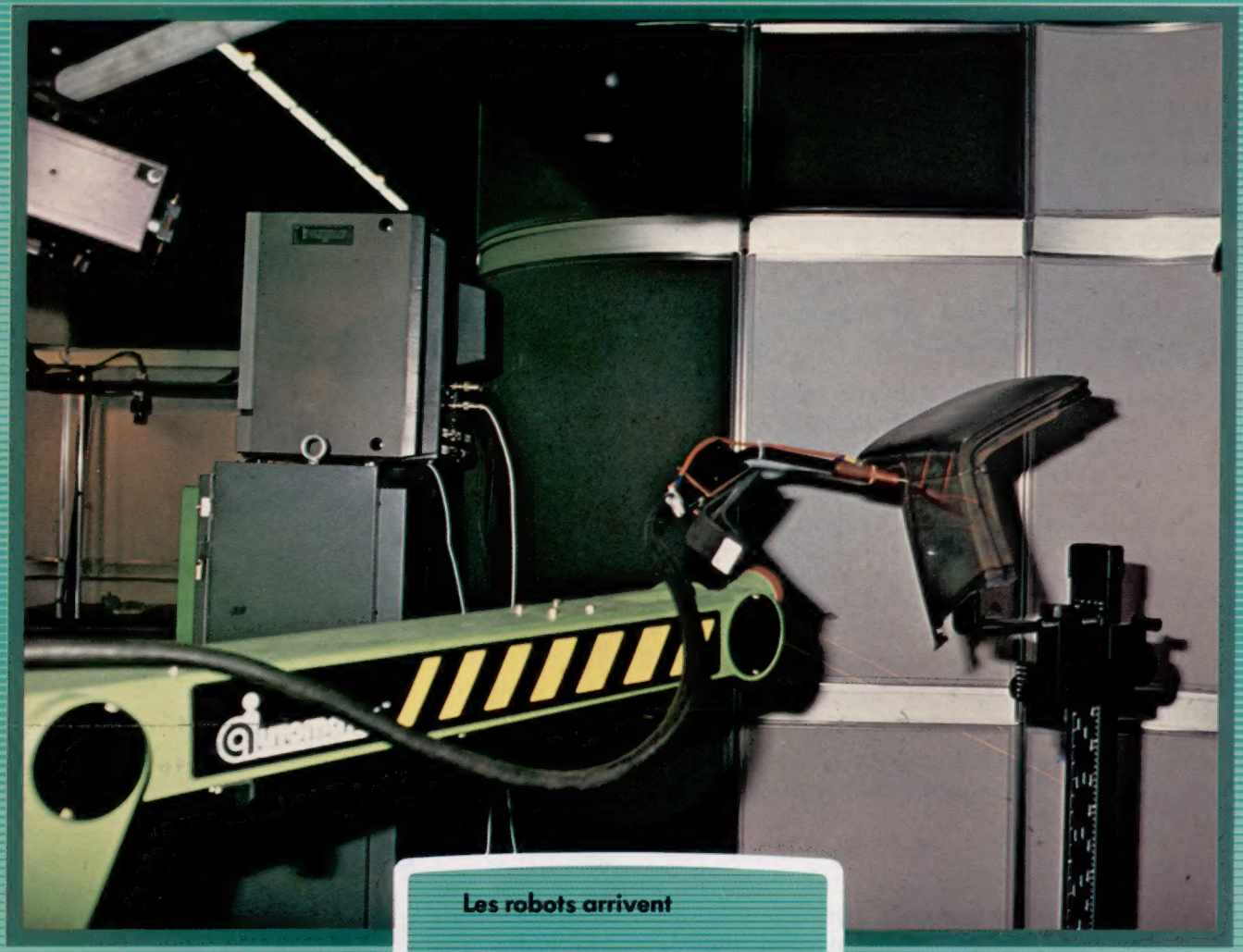
ABC INFORMATIQUE

N°55

COURS D'INFORMATIQUE PRATIQUE ET FAMILIALE



Les robots arrivent

Logo et récursivité

Contrôle de routine

Le PX-8 d'Epson

EDITIONS ATLAS

M6062-55-12-F 85FB-3,80FS-$1.95

Page manquante
(publicité et colophon)

Canards et androïdes

Pour explorer la science de la robotique, nous commençons par étudier les diverses tentatives visant à construire des robots, à partir du XVIIIe siècle jusqu'au robot industriel actuel.

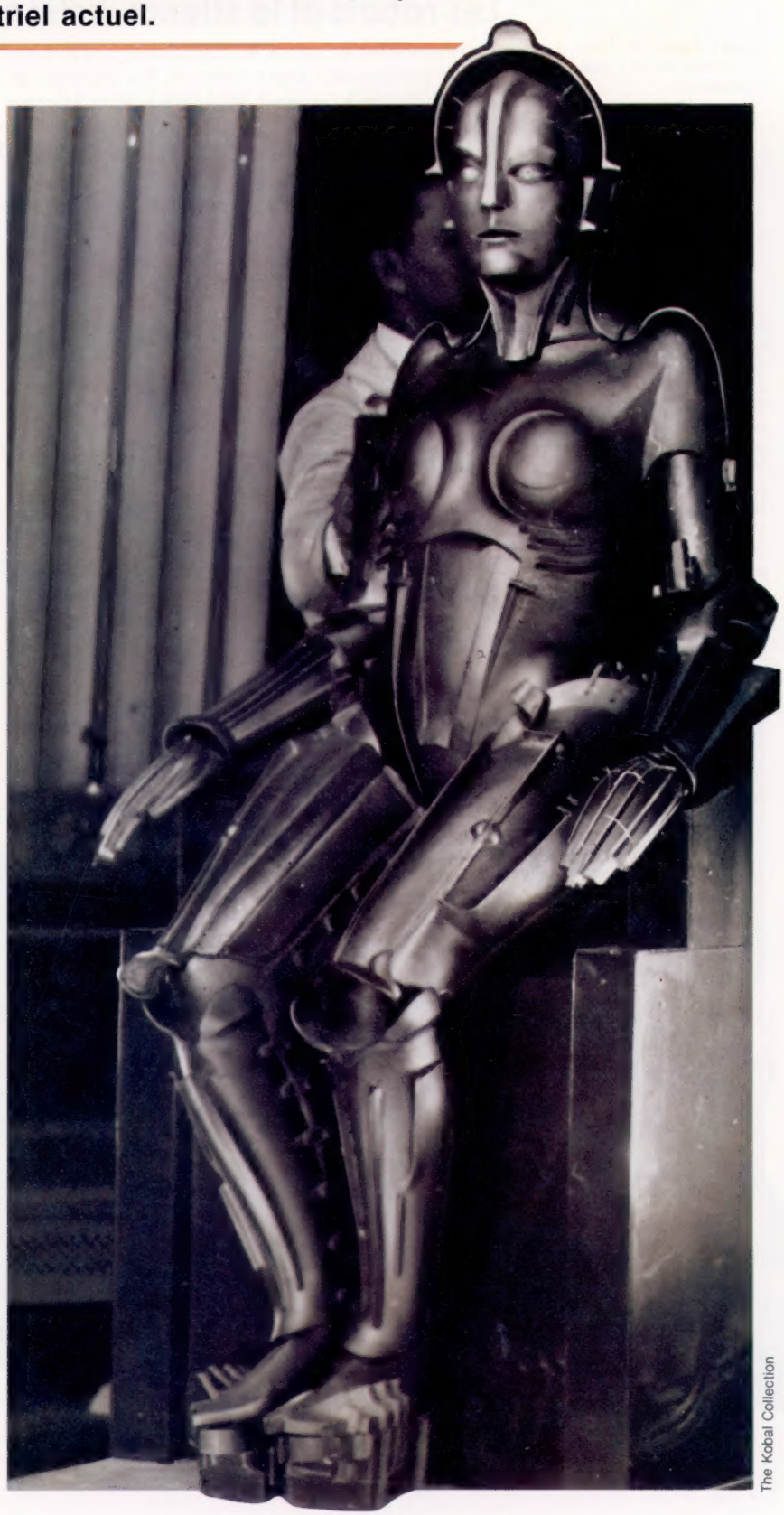
Depuis des centaines d'années, les automates fascinent de nombreuses personnes. Des philosophes, des ingénieurs et des inventeurs se sont employés à essayer de créer des machines qui pourraient imiter le comportement humain. Bien que de nos jours les robots ressemblent de moins en moins aux êtres humains et soient conçus de façon à accomplir une gamme spécifique de fonctions, les premiers « hommes mécaniques » devaient ressembler le plus possible à l'être humain, et on espérait qu'ils seraient en mesure d'accomplir toutes les tâches humaines.

Cependant, le premier robot mécanique ne prit pas une forme humaine. En 1738, Jacques de Vaucanson (1709-1782), un ingénieur français, présenta un canard mécanique à l'Académie royale des sciences de Paris. Le canard était capable d'agiter ses ailes, de cancaner et de manger. Plus tard, au XVIIIe siècle, un inventeur suisse, Pierre Jacquet-Droz (1721-1790), créa un ensemble de marionnettes mécaniques qui étaient à même d'effectuer diverses tâches. L'une pouvait écrire, une autre pouvait dessiner, et une autre pouvait jouer de l'orgue. A la fin du XIXe siècle, il existait de nombreuses machines de ce type, la conception de toutes ces machines étant basée sur des mécanismes d'horlogerie.

Au cours du XIXe siècle, de nombreux dispositifs à apparence humaine furent créés, tous étant basés sur l'horlogerie. En 1893, Georges Moore construisit un homme mécanique dont la force motrice était la vapeur — il pouvait entre autres choses fumer un cigare!

L'arrivée de nouvelles technologies a stimulé le développement de machines plus ambitieuses : des hommes mécaniques simples, construits à l'aide de pièces Meccano, capables de marcher, jusqu'à l'homme « Elektro » construit par la société américaine Westinghouse. Elektro était un homme mécanique de 2,15 m de hauteur, qui pouvait prononcer jusqu'à quatre-vingts mots différents, compter, saluer et établir une distinction entre diverses couleurs. Il était muni de onze moteurs électriques et pesait 117 kg! Un cerveau composé de quatre-vingt-deux relais différents commandait cet énorme dispositif.

Mais chacun de ces hommes mécaniques souffrait de certaines restrictions. Aucun de ces dispositifs, malgré l'intérêt évident qu'ils suscitaient, n'offrait le potentiel que l'on pouvait espérer d'un robot idéal. Un homme mécanique qui sait dessiner ne pourrait pas faire les courses à votre place, et un homme mécanique qui sait marcher d'un bout à l'autre d'une pièce ne pourrait pas



The Kobal Collection

Première star mécanique
Le classique du cinéma de science-fiction, *Metropolis* de Fritz Lang, influença les cinéastes et les spectateurs pendant des décennies, en cristallisant les vagues images de progrès et d'industrie sur La Machine, première star robot du cinéma.
Voir photo de la page précédente.

atteindre les boutiques sans heurter un lampadaire. Toutes ces représentations « humaines » n'étaient en fait que des machines — elles effectuaient une gamme limitée de tâches qui ne nécessitaient aucune prise de décision et n'exigeaient aucune forme d'intelligence.

Les robots et la science-fiction

Mais si les inventeurs et les ingénieurs n'arrivaient pas à concrétiser leurs espoirs, les auteurs de science-fiction n'avaient qu'à laisser aller le cours de leur imagination. Le mot « robot » tire son origine d'une œuvre de science-fiction. En 1923, l'auteur tchèque Karel Capek (1789-1938) écrivit une pièce qu'il intitula *Les Robots universels de Rossum*. La pièce parlait de l'invention d'hommes mécaniques d'une perfection telle qu'ils pouvaient se charger de toutes les tâches pouvant être effectuées par un être humain. Ces robots en vinrent à la conclusion qu'ils n'avaient plus besoin des hommes, ce qui plaça ces derniers dans une position plutôt délicate. En tchèque, le mot *robota* signifie simplement « travailleur ». Ainsi, le titre de la pièce de Capek aurait pu être traduit par *Les Travailleurs universels de Rossum;* mais le mot « robot » fut retenu et devint le terme utilisé pour désigner toute machine dotée de capacités humaines.

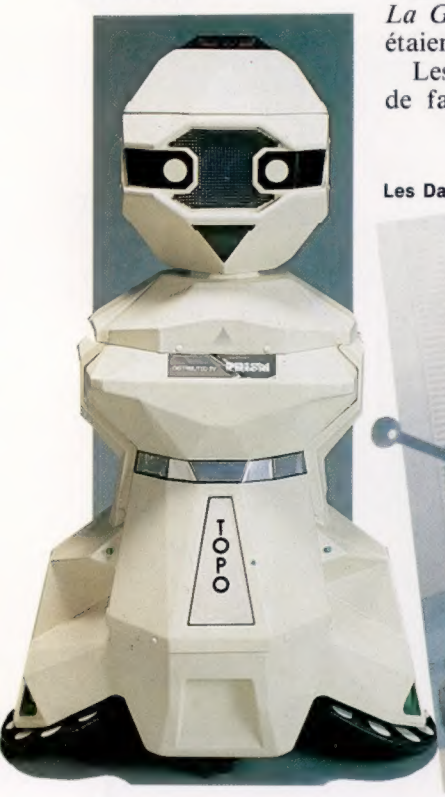
Les œuvres de fiction parlant de créatures construites à l'image de l'homme remontent au roman *Frankenstein,* de Mary Shelley (1818). Bien qu'il ne fût pas mécanique, le monstre créé par Victor Frankenstein était construit à partir d'un ensemble de pièces. Les envahisseurs dans *La Guerre des mondes* de H.G. Wells (1898) étaient également des robots.

Les romanciers du XX^e siècle ont exploré de façon détaillée l'univers fictif des robots. L'œuvre d'Isaac Asimov est sans doute la plus importante. Ce célèbre auteur de science-fiction commença sa carrière en 1940 en écrivant des nouvelles parlant de robots et de leurs problèmes de fonctionnement. L'univers robotique d'Asimov est si complet qu'il a même formulé les trois lois de la robotique. Selon Asimov, ces lois seront données dans *Le Manuel de robotique* (56^e édition, en 2058). Ce laps de temps raisonnable rend plausible un réel développement des robots.

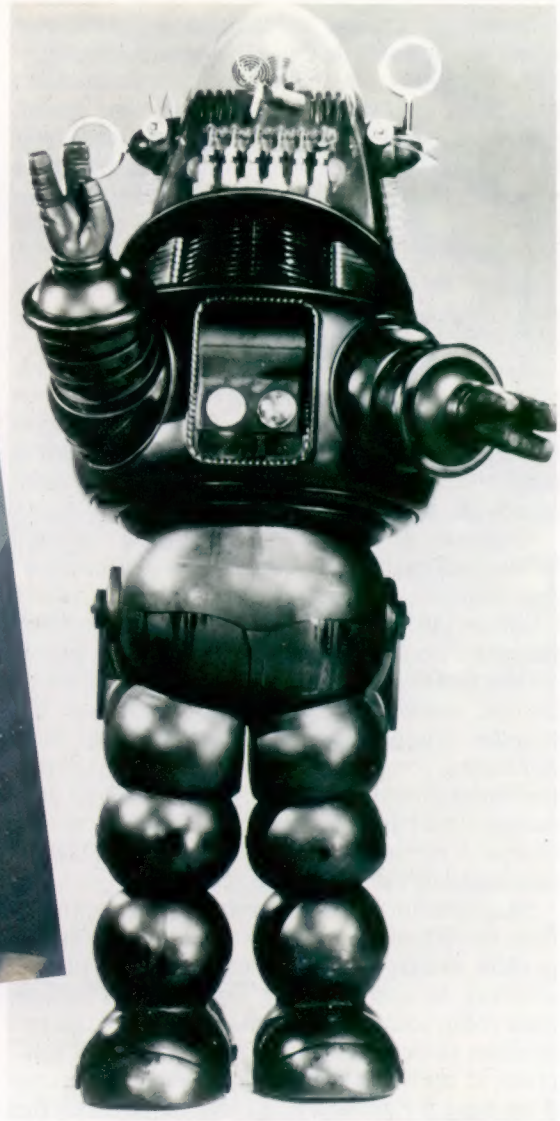
Les robots se sont aussi imposés au cinéma et à la télévision. La série télévisée *Doctor Who* était fortement peuplée de ce type de créatures, et, dans *Star Wars,* C3PO et R2D2 sont placés sur un pied d'égalité avec les autres stars humaines.

Comparée à cet univers fictif, l'utilisation actuelle des robots semble assez banale. On estime à vingt-cinq mille le nombre de robots qui seront utilisés à la fin de 1985 dans l'industrie japonaise, à quinze mille, aux États-Unis, et à huit mille en République fédérale d'Allemagne. L'expansion du parc robotique européen continuera de façon rapide, et on estime qu'en 1990 il représentera une valeur de trois milliards de francs.

Fait et fiction
Les robots les plus célèbres à la télévision sont sans doute les Daleks. Ce sont des transporteurs blindés personnels, commandés de l'intérieur par leurs créateurs.
Robbie, le robot du film *La Planète interdite,* incarne l'image du robot sensible et intelligent.
Prism, le robot domestique dont on a abandonné la production, était une tentative d'introduction de la robotique à la maison.



Les Daleks



Robbie, le robot

Mais, pour bien des gens, les robots industriels ne présentent aucun intérêt. Une machine qui soude des pièces de façon répétée sur un châssis d'automobile, ou peint au fusil, correspond difficilement à l'image que l'on s'est faite du robot. Le robot idéal existera-t-il un jour ? Un tel robot sera-t-il conçu à l'image de l'homme ? Tout cela reste à déterminer, mais, en examinant certains aspects de la robotique comme nous le ferons dans cette série d'articles, nous pourrons juger nous-mêmes de son avenir.

Langage de robots

Les robots, en fiction tout au moins, ont suscité la création d'une telle variété de noms que nous avons jugé utile de présenter un glossaire des termes les plus utilisés. Cependant, gardez à l'esprit qu'il ne suffit pas de nommer quelque chose pour que cela existe réellement !

Androïde : robot conçu à l'image de l'homme.

Anthropomorphique : « comme l'homme ». Un androïde est anthropomorphique à tous égards, mais de nombreux robots ne sont anthropomorphiques qu'à certains points de vue. Par exemple, un robot peut être doté d'un bras identique à celui d'un humain.

Automation : commande automatique d'un procédé de fabrication.

Automate : machine qui effectue généralement une série prédéterminée de fonctions. Les premiers hommes mécaniques étaient des automates. Ce terme a également une signification plus technique lorsqu'il est employé dans l'expression théorique de l'automate, qui désigne un système analytique, grâce auquel tout dispositif peut être étudié — robots, ordinateurs, même des personnes.

Cols métalliques : robots industriels. Les travailleurs de bureau sont appelés « cols blancs » et les travailleurs manuels « cols bleus ». C'est ainsi que les robots ont reçu le nom de « cols métalliques ».

Cybernétique : étude de systèmes de commande et de communication. Conçue par Norbert Weiner en 1947, la cybernétique a pour principal objectif d'examiner des systèmes biologiques comme s'il s'agissait de machines.

Manipulateur : terme désignant la « main » d'un robot.

Mécanisation : remplacement d'un processus manuel par un processus mécanique.

Robot : machine capable d'effectuer certaines tâches humaines, bien qu'elle ne doive pas nécessairement avoir la même apparence.

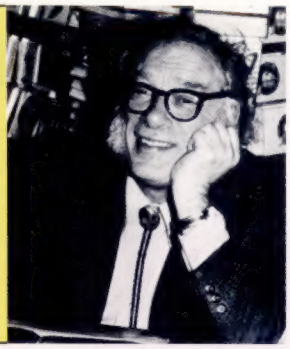
Robotique : science qui étudie les robots.

Les lois de robotique d'Asimov

1. Un robot ne doit pas blesser un être humain, ou, à la suite d'une non-intervention, lui causer une blessure.

2. Un robot doit obéir aux ordres provenant d'êtres humains, sauf si ces ordres violent la première loi.

3. Un robot doit protéger sa propre existence aussi longtemps que cette protection ne viole pas les deux premières lois.



Règlements pour robots
Lorsque les robots seront capables d'accomplir des tâches de façon indépendante, les lois d'Asimov pourront définir ce que doit être leur comportement.

Renault — chaîne Super 5

Assemblage par robot
Pour encore quelque temps, les robots ne seront utilisés que sur les chaînes d'assemblage. L'économie due à la production en série en fait des travailleurs idéaux. La spécialisation exige généralement que ces robots soient réduits à un ou deux bras munis de pinces et d'équipement de soudure. (Cl. G. Maillac/REA.)

Qu'est-ce qu'un robot?

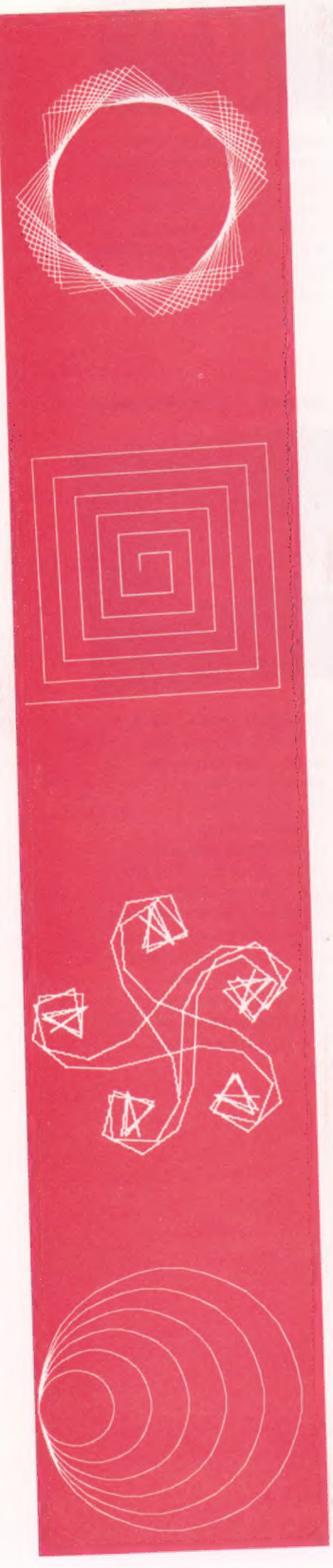
Si vous jetez un coup d'œil au glossaire, vous verrez que nous avons défini un robot comme étant « une machine capable d'effectuer toute tâche humaine, bien qu'elle ne doive pas nécessairement avoir une apparence humaine ». Évidemment cette définition est très large. En pratique cependant, un robot doit être doté de certaines qualités humaines reconnaissables. Il peut être capable de se déplacer, et même de marcher. Il peut posséder un bras qui ressemble à un bras humain. Il peut être capable d'entendre et de voir, et même être doté d'un degré d'intelligence assez élevé.

La forme et les capacités d'un robot dépendent principalement de deux choses : ce que nous voulons qu'il fasse, et ce que nous pouvons lui faire faire. Par exemple, il peut arriver qu'un robot industriel utilisé pour la soudure ne se déplace pas — non parce qu'un robot ne peut pas se déplacer, mais parce que nous désirons qu'il reste immobile pour se consacrer uniquement à la soudure. De même, un robot domestique peut être capable de faire une tasse de thé, mais il se peut qu'il ne soit pas capable de monter l'escalier pour vous l'apporter au lit, car comment construire un robot qui puisse monter des marches sans renverser votre thé ?

Le terme robot est devenu le mot générique qui désigne toutes les machines construites à l'image de l'homme, et les restrictions, quant à leur nature et à leur potentiel, dépendent de ceux qui les conçoivent et les construisent. Ces restrictions diminuent presque quotidiennement.

Répéter la scène

Le langage LOGO utilise la technique mathématique de la récursivité (une instruction qui renvoie à elle-même). Elle produit des résultats surprenants, notamment en conjonction avec des entrées variables.



Un des premiers programmes définis dans ce cours consistait à créer une procédure pour dessiner un carré. La tortue devait avancer, tourner de 90°, et répéter ces deux étapes trois autres fois. Voici une autre façon de tracer un carré :

```
POUR CARRÉ
    AV 50
    DR 90
    CARRÉ
FIN
```

Si vous essayez cette procédure, vous vous apercevrez que la tortue trace un carré presque complet, puis en recommence un autre à la périphérie du premier. Elle continue ainsi en une sorte de spirale carrée, jusqu'à ce que vous fassiez CONTROL-G ou BREAK. Notons que cette nouvelle procédure s'appelle elle-même : elle est récursive.

Lorsque la procédure est lancée, LOGO cherche la définition de CARRÉ et commence à exécuter les instructions. La tortue AVance de 50 unités et tourne à DRoite de 90°. La prochaine instruction est CARRÉ, aussi LOGO cherche la définition de CARRÉ et lui obéit : la boucle s'enchaîne sur elle-même. Sans intervention, cela peut se poursuivre à l'infini.

Il est également possible d'utiliser des appels récursifs dans des procédures qui nécessitent des entrées :

```
POUR POLY:CÔTÉ :ANGLE
    AV :CÔTÉ
    DR :ANGLE
    POLY :CÔTÉ :ANGLE
FIN
```

Cette procédure peut tracer tous les polygones définis jusqu'ici dans le cours, ainsi que beaucoup d'autres que nous n'avons pas vus (pourquoi ne pas essayer un angle de 89° ?). Il est également possible de modifier la valeur des données dans l'appel récursif, ainsi :

```
POUR POLYSPI :CÔTÉ :ANGLE
    AV :CÔTÉ
    DR :ANGLE
    POLYSPI ( :CÔTÉ + 5) :ANGLE
FIN
```

La seule différence entre cette procédure SPIrale et la procédure précédente POLYgone est que la valeur pour CÔTÉ est incrémentée de 5 à chaque nouvel appel. Aussi, si vous commencez avec POLYSPI 10 90, le premier appel tracera une ligne de 10, le deuxième de 15, le troisième de 20, etc. Le résultat sera une spirale évasée. Vous pouvez essayer avec différentes données en entrée : 10 90, 10 95, 10 120, 10 117, 10 144 et 10 142 sont de bonnes valeurs de départ. Vous pouvez aussi essayer de modifier la procédure, par exemple en remplaçant l'addition par une soustraction ou par une multiplication.

Voici une autre procédure qui incrémente l'angle à la place du côté :

```
POUR INSPI :CÔTÉ :ANGLE :INC
    AV :CÔTÉ
    DR :ANGLE
    INSPI :CÔTÉ ( :ANGLE + :INC) : INC
FIN
```

Essayez différentes données : 5 0 7, 10 40 30, 15 2 20, 5 30 20 seront un bon début. Pourquoi certaines formes se ferment-elles sur elles-mêmes et d'autres non ? Pouvez-vous dégager une règle à ce sujet ?

Une répétition d'un fragment de code est appelée « itération ». LOGO utilise RÉPÈTE à cette fin, alors que d'autres langages utilisent FOR...NEXT, REPEAT...UNTIL et WHILE...WEND, etc. Cependant, LOGO est beaucoup plus basé sur la récursivité que sur l'itération. Si vous avez souvent programmé dans d'autres langages, vous éprouverez peut-être de la difficulté à vous passer des itérations, mais le graphique de la tortue est idéal pour faire l'expérience des appels récursifs de procédures.

Règles d'arrêt

Les procédures que nous avons vues jusqu'à présent se poursuivaient indéfiniment. Il nous faut donc trouver un moyen d'arrêter une procédure à un moment donné. En reprenant l'exemple de la procédure CARRÉ, un arrêt interviendrait après un cycle complet (un carré entier) lorsque POSITION=0. Cela peut être obtenu en ajoutant une règle d'arrêt à la procédure :

```
POUR CARRÉ :CÔTÉ
    AV :CÔTÉ
    DR 90
    SI POS = 0 ALORS STOP
    CARRÉ :CÔTÉ
FIN
```

Les nouvelles primitives sont STOP et SI. La première suscite une interruption de procédure et redonne le contrôle à la procédure d'appel. SI est la manière LOGO de prendre des décisions. SI est suivi d'une condition ; ALORS, d'une action qui est entreprise lorsque la condition est vraie.

Examinons de près une version de POLYSPI avec règle d'arrêt, pour voir ce qui se passe :

```
POUR POLYSPI :LONGUEUR
    SI :LONGUEUR > 15 ALORS STOP
    AV :LONGUEUR
    DR 90
    POLYSPI ( :LONGUEUR + 5)
FIN
```

Voici ce qui se passe lors de l'exploitation de POLYSPI 10 : la procédure est appelée, et une variable locale est définie avec pour valeur 10. Puisque cette valeur n'est pas supérieure à 15, LOGO continue avec AV 10 DR 90; il fait un nouvel appel à POLYSPI, mais cette fois avec une valeur en entrée de 15. La procédure est de nouveau appelée. Du fait que LONGUEUR n'est pas supérieur à 15, la tortue avance selon AV 15 DR 90, avec pour résultat d'appeler à nouveau POLYSPI. Mais, cette fois, la variable est passée à 20, aussi la procédure s'arrête-t-elle et redonne le contrôle à la procédure d'appel (POLYSPI 15). Cette procédure est arrivée à son terme et redonne le contrôle à sa propre procédure d'appel. Cette dernière s'arrête également, et le programme est donc terminé.

Nous avons vu que, dans LOGO, la « récursivité » exprime des procédures appelant leurs propres doubles. Il est important de ne pas oublier que, dans LOGO, les appels récursifs sont des copies des procédures d'origine, mais qui existent indépendamment d'elles. Lorsqu'une de ces copies s'arrête, elle redonne invariablement le contrôle à la procédure qui l'a appelée. Afin d'illustrer ce principe, nous pouvons retravailler POLYSPI de la sorte :

```
POUR POLYSPI :LONGUEUR
    SI :LONGUEUR > 15 ALORS STOP
    POLYSPI ( :LONGUEUR + 15)
    AV :LONGUEUR
    DR 90
FIN
```

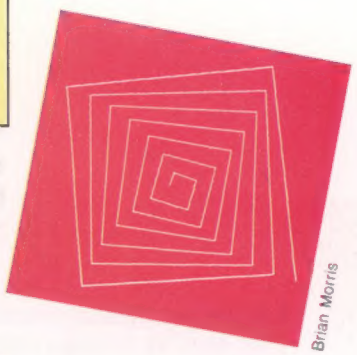
Si vous essayez ce code, vous verrez que le programme dessine à l'envers : la spirale va vers l'intérieur au lieu de l'extérieur (cela apparaîtra plus nettement si vous utilisez une valeur plus importante pour l'instruction de condition — par exemple 50 au lieu de 15). Ce qui est important ici est que LOGO dessine une ligne lorsque le contrôle revient d'un appel de procédure.

Dans un précédent exemple, une ligne était dessinée puis le contrôle passait à une autre procédure. Mais, ici, toutes les procédures sont appelées avant le commencement du dessin, et la dernière valeur créée pour LONGUEUR est la première utilisée dans le tracé.

Il faut noter en dernier lieu que la récursivité est une technique qui utilise beaucoup de mémoire. Cependant, les procédures où l'appel récursif figure sur la dernière ligne sont les plus efficaces — dans la mesure où elles ne nécessitent pas de mémoire supplémentaire, quel que soit le nombre d'appels. On dit d'une telle procédure qu'elle est « récursive par la fin », et il faudra toujours essayer d'y parvenir.

Problème n° 3 de procédure

Écrivez une procédure récursive pour tracer une tour de carrés empilés, en diminuant de moitié, à chaque étage, la longueur du côté.

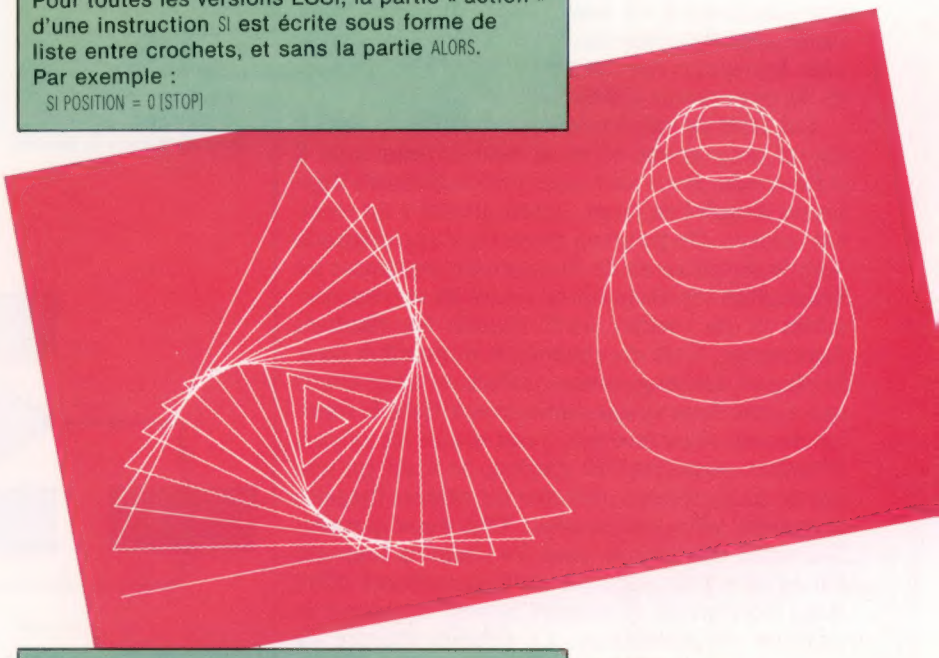


Brian Morris

Nuances logo

Pour toutes les versions LCSI, la partie « action » d'une instruction SI est écrite sous forme de liste entre crochets, et sans la partie ALORS.
Par exemple :

```
SI POSITION = 0 [STOP]
```



Réponses aux exercices

Procédure pour tracer un cercle, le rayon étant une entrée de données, avec la position présente comme un point de la circonférence.

```
POUR CERCLE :RAYON
    REPÈTE 36 [AV(2* :PI* :RAYON/36)
    DR 10]
FIN
FAIRE PI "PI (3,14159)
```

Si nous adaptons cette procédure pour que le centre du cercle soit la position présente, nous obtenons :

```
POUR C.CERCLE :RAYON
    CL 6H 90 AV :RAYON DR 90 CB
    CERCLE :RAYON
    CL 6H AR :RAYON DR 90 CB
FIN
```

CIBLE utilise C.CERCLE pour tracer 5 cercles concentriques :

```
POUR CIBLE
    C.CERCLE 10 C.CERCLE 20 C.CERCLE 30
    C.CERCLE 40 C.CERCLE 50
FIN
```

Contrôle de routine

Nos articles sur les techniques de programmation devraient vous avoir donné beaucoup d'idées sur la conception et le développement des programmes. Abordons cette fois les méthodes de test.

Un des principaux avantages de la programmation dans un langage interprété comme le BASIC est que le code peut être testé en même temps qu'il est écrit. Le programmeur peut à tout moment taper RUN et voir ce qui se passe. Sur la plupart des machines, il est assez simple de s'introduire dans un programme en cours d'exécution, d'afficher les valeurs des variables clés, de les changer et de poursuivre (CONtinue).

Les tests de validation sont destinés à vérifier qu'un programme effectue bien ce pour quoi il a été conçu. Pour tout ensemble de données légales ou illégales en saisie, il doit produire un résultat correct. On pourrait penser qu'une méthode simple de test serait de fournir au programme un échantillon représentatif de toutes les entrées possibles et de vérifier qu'il fournit les résultats appropriés. Cela est cependant impossible pour la plupart des programmes. Même un programme qui ajoute deux valeurs entières et affiche le résultat devrait alors être testé pour toutes les valeurs d'entiers possibles ! Et encore, ce n'est qu'une partie du problème, puisque les valeurs illégales devraient aussi être testées ! Une autre possibilité serait d'examiner tous les chemins logiques du programme. Chaque embranchement offre deux possibilités et chaque boucle augmente le nombre de possibilités. Le schéma montre un programme simple avec une boucle comportant des instructions IF...THEN. Il y a quatre chemins dans la boucle et la boucle est exécutée dix fois. Ce qui signifie que le nombre de chemins distincts depuis le début jusqu'à la fin du programme est de 1 398 100 — un nombre incroyable pour un code d'une douzaine de lignes seulement. Impossible de tout tester !

Alors, si ni le test complet des données ni le test exhaustif des chemins logiques ne peuvent être mis en œuvre, que faire ? Il n'existe pas de manières infaillibles de tester un programme d'une complexité moyenne en un temps plausible. Les tests obéissent à la loi suivante : le nombre d'erreurs trouvées par unité de temps consacrée aux tests décroît avec chaque nouvelle unité de temps supplémentaire. Aussi considère-t-on qu'il faut arrêter les tests lorsque le temps qu'on y passe coûte plus cher que les erreurs trouvées dans le programme.

Pourtant, malgré ces inconvénients, il est nécessaire de mettre au point une méthode de tests. On peut partir du postulat que si une machine fonctionne correctement avec un élément de données d'un certain type, elle fonctionnera également bien avec tous les autres éléments du même type. Si un sous-programme marche avec un entier positif dans les valeurs admises, il marchera avec tous les entiers positifs de cet intervalle. Cela nous amène à écrire un test connu comme le « test des classes équivalentes ». Il s'agit de développer un ensemble de cas-tests qui sont tous représentatifs d'un ensemble de cas devant se comporter de la même façon. Si une partie de code doit vérifier qu'une entrée appartient à l'écart 1-100, nous effectuerons un test pour les entrées inférieures à 1, comprises entre 1 et 100 et, enfin, supérieures à 100 (valeur< 1, 1 =< valeur =< 100, valeur > 100).

Il est également possible de simplifier les chemins logiques en prenant chaque entrée de routine et en passant en revue tous les résultats possibles des embranchements de décision. Le schéma 2 comprend une routine de calcul des bonus de points d'un jeu. Avec les paramètres d'entrée BONUS, NIVEAU et COUPS, cette routine

Quatre en mains

Même une structure aussi simple que cette boucle ne peut être testée de manière absolue. Cela provient de la multiplicité des conditions en entrée : il existe plus d'un million de chemins logiques possibles dans la boucle.

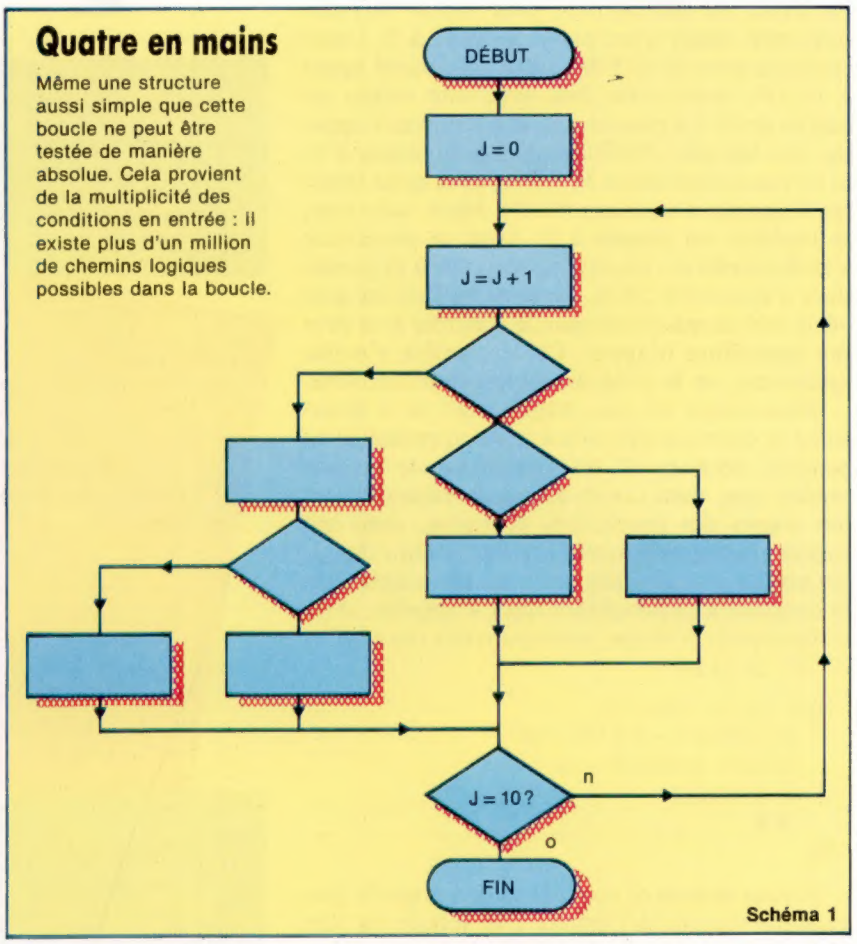


Schéma 1

Seulement un test
Un jeu complet de données-tests, calculé pour l'exemple illustré par l'organigramme, pourra se présenter ainsi :

ENTRÉE		SORTIE	
NIVEAU	COUPS	BONUS	BONUS
6	10	200	1300
4	10	550	2300
7	10	550	3950
4	10	200	800
7	10	200	1400
1	20	2500	2600
1	20	550	550
6	5	200	300
6	50	200	300
4	5	2500	2600
7	50	2500	2600
4	50	550	550
7	5	550	550

donne une nouvelle valeur à BONUS. Elle peut s'écrire de la sorte :

```
6030 IF NIVEAU > 2 AND COUPS = 10 THEN
     BONUS = BONUS*NIVEAU
6040 IF NIVEAU = 6 OR BONUS > 2000 THEN
     BONUS = BONUS + 100
```

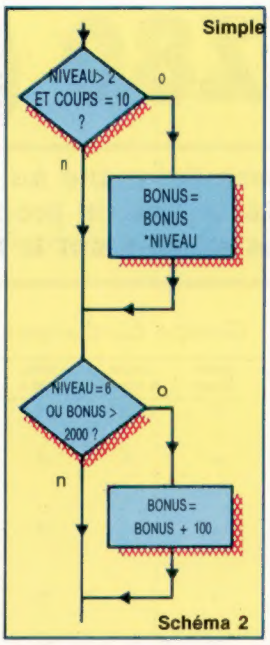
Pour passer en revue tous les résultats des expressions conditionnelles, nous devons envisager toutes les entrées (à ces expressions) qui donnent comme résultat « oui », ou « non ». Pour les deux résultats, nous nous intéressons aux effets produits par deux variables combinées par un opérateur logique (ET et OU). Cela signifie que nous devons prendre en compte les valeurs combinées des variables et non leurs valeurs individuelles. Pourquoi? Prenons comme exemple le cas où nous testons 4 et 1 pour NIVEAU, et 10, 5 et 20 pour COUPS dans la première décision. Lorsque NIVEAU=4, les trois valeurs de COUPS sont testées mais, par contre, lorsque NIVEAU=1, elles ne le sont pas. C'est un cas où une partie d'une décision masque une autre partie de la même décision. Pour tester chaque partie séparément, il est préférable de simplifier les décisions complexes.

Le schéma 3 nous montre qu'avec quatre décisions binaires il y a 2^4 (= 16) possibilités de résultat, et nous devons les envisager toutes. Recensons les conditions pour un résultat « oui » ou « non », pour chaque décision comme ci-après :

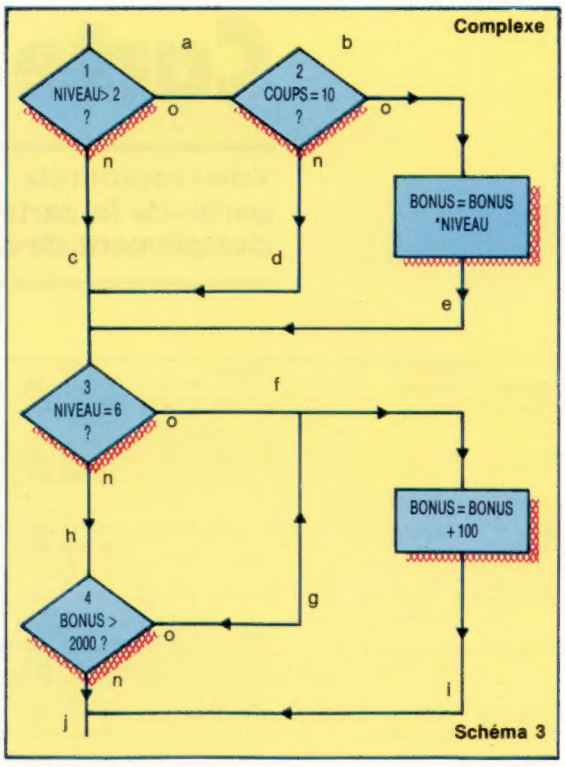
	1	2	3	4
oui	NIVEAU> 2	COUPS = 10	NIVEAU = 6	BONUS> 20000
non	NIVEAU = 2 NIVEAU< 2	COUPS< 10 COUPS> 10	NIVEAU< 6 NIVEAU> 6	BONUS = 20000 BONUS< 20000

Cela peut servir à déterminer les valeurs pour des données de test représentatives. Par exemple, pour le chemin logique a-d-f-i (voir le schéma 3), NIVEAU doit être supérieur à 2 ou égal à 6, COUPS ne doit pas être égal à 10, et BONUS peut prendre n'importe quelle valeur (elle n'est pas en cause). Les valeurs NIVEAU=6, COUPS=20 et BONUS=150 seraient conformes à ce chemin logique, comme bien d'autres, bien sûr. Le chemin logique a-b-e-h-j serait testé avec NIVEAU=4, COUPS=10 et BONUS=600 (n'oubliez pas que nous parlons de la valeur d'entrée de BONUS qui est ensuite susceptible d'être multipliée par NIVEAU). Il convient aussi de calculer les résultats produits par chaque échantillon de données *avant* d'effectuer le test, pour pouvoir comparer les résultats. Les données de saisie elles-mêmes testeront le fonctionnement ou le non-fonctionnement du programme. Pour savoir si le programme fait bien ce qu'il doit faire, le résultat doit être calculé au préalable (à la main). Un jeu complet de cas-tests pour cet exemple est donné à gauche.

Dotés d'une méthode d'examen de la validité du logiciel, il nous faut maintenant pouvoir traiter un grand programme sans que la complexité qui en résulte soit exorbitante. Nous voyons à



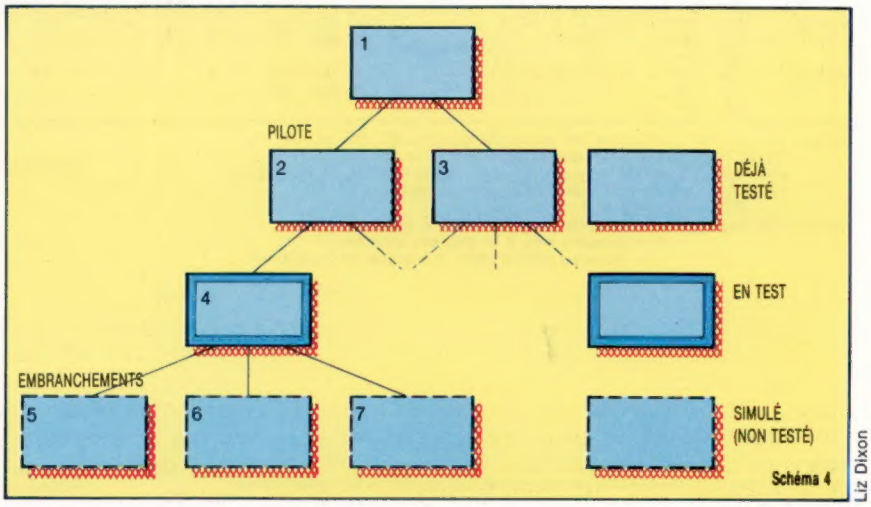
Masquer une décision
Attribuer une lettre distinctive à chaque transition rend les tests plus faciles.



cette occasion une autre application de la programmation structurée. Les programmes écrits sous forme d'un ensemble de modules indépendants, ordonnés hiérarchiquement, nous permettent de tester chaque module individuellement. Nous pouvons commencer par le module le plus haut et poursuivre vers le bas. Nous ne passons au module suivant que lorsque tous les modules en amont ont été testés. En outre, nous pouvons utiliser les modules précédents pour fournir des données aux modules situés plus bas.

Le module en cours de test a, au-dessus de lui (à moins qu'il soit le premier), un module *pilote* entièrement testé. Les modules suivants sont des branches non testées et donc non fiables. Aussi sont-elles simulées par de courts codes qui renvoient simplement les données de test appropriées, lorsqu'ils sont appelés par le module en cours de test. Le schéma 4 montre le principe. Les modules 1, 2 et 3 ont déjà été testés, et les modules 5, 6 et 7 sont simulés.

Tests d'amont en aval
Les tests se trouvent grandement simplifiés par cette approche. En effet, chaque module est testé dès qu'il est écrit, isolément et en association avec d'autres modules testés. Le comportement de ceux non encore écrits peut être simulé par l'écriture de « codes-embranchements » qui donnent artificiellement des exemples de résultats attendus.



Liz Dixon

Carte Z80 (suite)

Voici reproduite, avec l'aimable autorisation de Zilog Inc., la seconde partie de la carte référence du programmeur Z80, qui vient en complément de nos articles sur le langage machine.

Groupe de chargement 16 bits

Mnémonique	Opération symbolique	S	Z	Drapeaux H	P/V	N	C	Opc 76 543 210	Hex	Nombre d'octets	Nombre de cycles M	Nombre d'états T	Commentaires
LD dd, nn	dd ← nn	•	•	X • X	•	•	•	00 dd0 001 ← n → ← n →		3	3	10	
LD IX, nn	IX ← nn	•	•	X • X	•	•	•	11 011 101 00 100 001 ← n → ← n →	DD 21	4	4	14	
LD IY, nn	IY ← nn	•	•	X • X	•	•	•	11 111 101 00 100 001 ← n → ← n →	FD 21	4	4	14	
LD HL, (nn)	H ← (nn + 1) L ← (nn)	•	•	X • X	•	•	•	00 101 010 ← n → ← n →	2A	3	5	16	
LD dd, (nn)	dd_H ← (nn + 1) dd_L ← (nn)							11 101 101 01 dd1 011 ← n → ← n →	ED	4	6	20	
LD IX, (nn)	IX_H ← (nn + 1) IX_L ← (nn)	•	•	X • X	•	•	•	11 011 101 00 101 010 ← n → ← n →	DD 2A	4	6	20	
LD IY, (nn)	IY_H ← (nn + 1) IY_L ← (nn)	•	•	X • X	•	•	•	11 111 101 00 101 010 ← n → ← n →	FD 2A	4	6	20	
LD (nn), HL	(nn + 1) ← H (nn) ← L	•	•	X • X	•	•	•	00 100 010 ← n → ← n →	22	3	5	16	
LD (nn), dd	(nn + 1) ← dd_H (nn) ← dd_L	•	•	X • X	•	•	•	11 101 101 01 dd0 011 ← n → ← n →	ED	4	6	20	
LD (nn), IX	(nn + 1) ← IX_H (nn) ← IX_L	•	•	X • X	•	•	•	11 011 101 00 100 010 ← n → ← n →	DD 22	4	6	20	
LD (nn), IY	(nn + 1) ← IY_H (nn) ← IY_L	•	•	X • X	•	•	•	11 111 101 00 100 010 ← n → ← n →	FD 22	4	6	20	
LD SP, HL	SP ← HL	•	•	X • X	•	•	•	11 111 001	F9	1	1	6	
LD SP, IX	SP ← IX	•	•	X • X	•	•	•	11 011 101 11 111 001	DD F9	2	2	10	
LD SP, IY	SP ← IY	•	•	X • X	•	•	•	11 111 101 11 111 001	F9 F9	2	2	10	
PUSH qq	(SP − 2) ← qq_L (SP − 1) ← qq_H SP → SP − 2	•	•	X • X	•	•	•	11 qq0 101		1	3	11	
PUSH IX	(SP − 2) ← IX_L (SP − 1) ← IX_H SP → SP − 2	•	•	X • X	•	•	•	11 011 101 11 100 101	DD E5	2	4	15	
PUSH IY	(SP − 2) ← IY_L (SP − 1) ← IY_H SP → SP − 2	•	•	X • X	•	•	•	11 111 101 11 100 101	FD E5	2	4	15	
POP qq	qq_H ← (SP + 1) qq_L ← (SP) SP → SP + 2	•	•	X • X	•	•	•	11 qq0 001		1	3	10	
POP IX	IX_H ← (SP + 1) IX_L ← (SP) SP → SP + 2	•	•	X • X	•	•	•	11 011 101 11 100 001	DD E1	2	4	14	
POP IY	IY_H ← (SP + 1) IY_L ← (SP) SP → SP + 2	•	•	X • X	•	•	•	11 111 101 11 100 001	FD E1	2	4	14	

dd	Paire
00	BC
01	DE
10	HL
11	SP

qq	Paire
00	BC
01	DE
10	HL
11	AF

Notes : dd est l'une quelconque des paires de registres BC, DE, HL, SP.
qq est l'une quelconque des paires de registres AF, BC, DE, HL.
$(PAIRE)_H$ et $(PAIRE)_L$ renvoient à l'octet respectivement le plus significatif et le moins significatif de la paire de registres.
Ex. : BC_L = C; AF_H = A.

Notation de drapeaux : • = drapeau non affecté, 0 = drapeau à zéro,
1 = drapeau mis, X = drapeau inconnu,
↕ = drapeau affecté selon le résultat de l'opération.

NOTE :
Les instructions d'entrée et sortie de pile (PUSH et POP) ajustent le SP après chaque exécution.

		SOURCE									
		REGISTRE							IMM. EXT.	ADR. EXT.	REG. INDIR.
		AF	BC	DE	HL	SP	IX	IY	nn	(nn)	(SP)
DESTINATION	AF										F1
	BC								01 n n	ED 4B n n	C1
	DE								11 n n	ED 5B n n	D1
	HL								21 n n	2A n n	E1
	SP				F9		DD F9	FD F9	31 n n	ED 7B n n	
	IX								DD 21 n n	DD 2A n n	DD E1
	IY								FD 21 n n	FD 2A n n	FD E1
	ADRESSE EXTERNE (nn)		ED 43 n n	ED 53 n n	22 n n	ED 73 n n	DD 22 n n	FD 22 n n			
PUSH INSTRUCTIONS	REGISTRE IND. (SP)	F5	C5	D5	E5		DD E5	FD E5			

Poids plume

On appelle « ordinateurs portables » des appareils très différents. Certains tiennent dans une valise. Parmi les plus récents, nous présentons ici le PX-8 d'Epson.

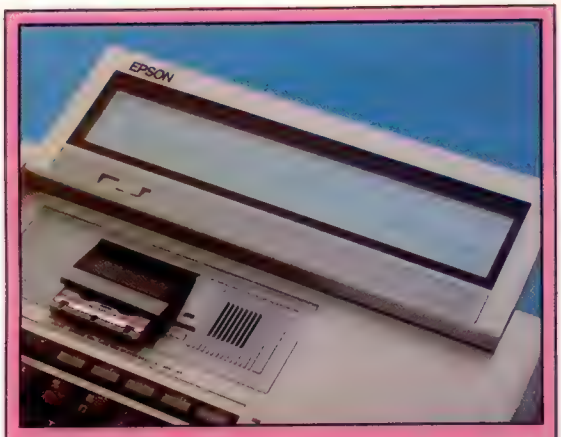
Le PX-8 a les dimensions d'un annuaire téléphonique, et pèse environ 2,300 kg. Le boîtier est de couleur beige (de deux tons), et comporte une poignée métallique repliable. Le tout ne fait guère penser à un ordinateur. Pourtant, une partie amovible révèle un clavier complet, de type machine à écrire, et un écran d'affichage qu'on libère en appuyant sur un bouton-poussoir nommé « UNLOCK ». Cela permet également d'accéder à un lecteur de microcassettes. L'écran lui-même est mobile, et peut prendre onze positions différentes : mais, dans les faits, cinq ou six seulement offrent un bon angle de vue.

Le clavier AZERTY est composé de soixante-douze touches, de couleur différente suivant leur fonction. Les touches alphanumériques sont brun foncé. Tous les caractères « internationaux » peuvent être obtenus par manipulation des broches du boîtier de connexion : le manuel de l'utilisateur explique très clairement comment faire. Il existe également quatre touches de contrôle du curseur orange vif, trois touches système « ESCAPE », « PAUSE » et « HELP », et trois touches d'édition « INSERT », « DELETE » et « HOME ».

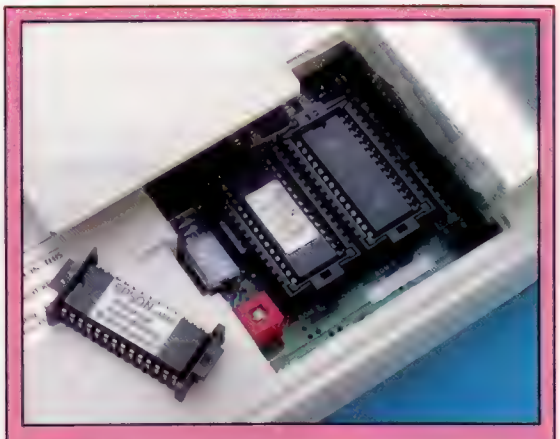
Le clavier est d'excellente qualité et d'usage facile, surtout lorsqu'on s'en sert en position inclinée. Si le PX-8 est posé sur un bureau, les choses deviennent un peu compliquées, car une forte pression sur les touches s'avère nécessaire. Le constructeur fournit deux pattes rétractables, qui malheureusement ne résolvent pas tout à fait le problème. L'appareil a plusieurs modes d'affichage : on peut passer de l'un à l'autre par appui des touches « INSERT », « CAPS LOCK » et « NUMBER », trois petites diodes de couleur rouge permettent à tout moment de savoir où on en est.

La documentation est exhaustive. Elle tient en deux manuels très épais. Le premier fait plusieurs centaines de pages, et traite des problèmes d'installation et de mise en route, de l'emploi du maté-

Dernier cri
Le PX-8 est fabriqué par Epson, à qui on doit déjà des imprimantes matricielles, ainsi que le HX-20, un portable, et le QX10, un appareil destiné à la gestion.
Il est pourvu de 64 K de RAM, d'un affichage à cristaux liquides, du système d'exploitation CP/M et de plusieurs logiciels intégrés.



Grand écran
Le PX-8 comporte un écran à cristaux liquides, alimenté par la pile principale de l'appareil. Cet écran offre une résolution de 480 × 64 pixels pour les représentations graphiques.



Des ROM multiples
En déplaçant un petit panneau situé en dessous du PX-8, on met à jour les prises sur lesquelles viennent s'enficher les logiciels installés sur ROM. Wordstar est intégré à la machine.

riel et des logiciels, ainsi que de la mise en œuvre du système CP/M. Il comporte aussi des cartes des zones mémoire, la liste complète de tous les caractères disponibles (avec leurs codes), plus un programme, assez long, en langage machine, grâce auquel on peut sauvegarder et charger des graphismes à partir d'une disquette. Le second est un magnifique guide de référence pour le BASIC. Il est aussi important que le précédent, et explique d'abord comment installer et utiliser le BASIC lui-même (qui n'est pas résident, mais gravé sur une « capsule » de ROM, comme les logiciels intégrés). Après quoi, il évoque les divers aspects de la programmation.

Le PX-8 fait usage d'une unité centrale compatible avec le Z80. Les puces sont de type CMOS (« Complementary Metal Oxide Semiconductor »), ce qui signifie qu'elles consomment beaucoup moins d'énergie; l'écran étant à cristaux liquides basse puissance, l'appareil tout entier peut très bien fonctionner sur piles. Il en faut deux : une pour l'alimentation principale, et la seconde pour assurer la sauvegarde des données, quand l'ordinateur est hors tension. A noter que la pile principale doit être chargée avant qu'on puisse se servir de l'appareil : la toute première fois, cela contraindra donc à un délai de 8 heures avant qu'on commence pour de bon. Cette pile est rechargeable, et permet un maximum de quinze heures d'opérations ininterrompues.

Une fois le PX-8 prêt à fonctionner, il faut d'abord procéder à l'initialisation du système d'exploitation. Le manuel explique très bien la marche à suivre : entrer la date, le jour, l'heure, procéder à diverses besognes de gestion interne. C'est ainsi que l'appareil peut réserver une partie de la RAM, et traitera désormais cette zone mémoire comme s'il avait affaire à un véritable lecteur de disquettes. Mais si l'utilisateur peut en déterminer lui-même les dimensions (de 9 à 24 K), il doit préalablement la formater, s'il veut qu'elle puisse stocker des données. Epson propose par ailleurs une unité extérieure, qui contient 120 K de mémoire supplémentaire.

Ces opérations étant menées à bien, le PX-8 charge le système d'exploitation CP/M à partir de la ROM, puis affiche un répertoire à l'écran : utilitaires CP/M, logiciels intégrés. Le chargement peut s'effectuer, en règle générale, à partir d'une cassette, d'une disquette ou de la ROM. Dans ce dernier cas, il s'agit de programmes contenus dans une puce EPROM qui se fixe sur une prise située en dessous de la machine. Tout comme l'interpréteur BASIC, les quatre logiciels fournis à l'achat sont ainsi sur « capsules » de ROM, et, pour en sélectionner un, il suffit de faire usage des touches curseur, puis d'appuyer sur RETURN. L'ordinateur procède alors au chargement, de la ROM (qui ici correspond aux lecteurs de disquettes A et B) à la RAM (lecteur de disquettes A).

L'écran à cristaux liquides permet un affichage de huit lignes de quatre-vingts caractères, et sa résolution graphique est de 480 × 64 pixels. Un gros inconvénient — le seul, à vrai dire, de cet appareil remarquable : si l'affichage des caractères tapés à l'écran est assez rapide, il faut beaucoup plus de temps pour les effacer.

L'éventail de programmes est très complet : un traitement de texte (Portable Wordstar), un tableur (Portable Calc), une base de données (Portable Scheduler), à quoi viennent s'ajouter un logiciel de télécommunications (à utiliser avec un modem) et un autre qui permet le transfert des données sur des machines plus grosses, comme le QX10 du même constructeur. Le PX-8, étant équipé du CP/M, a par ailleurs accès à bien des possibilités.

Le BASIC de l'appareil est de type Microsoft, enrichi de diverses améliorations, comme une commande AUTO qui permet la renumérotation automatique des lignes du programme. Citons aussi un éditeur d'écran très complet, des commandes graphiques et sonores, des instructions assurant la communication avec l'extérieur via une interface RS232 intégrée, et des commandes traitant l'unité de microcassettes comme si elle était un lecteur de disquettes (pour le stockage des données en accès direct).

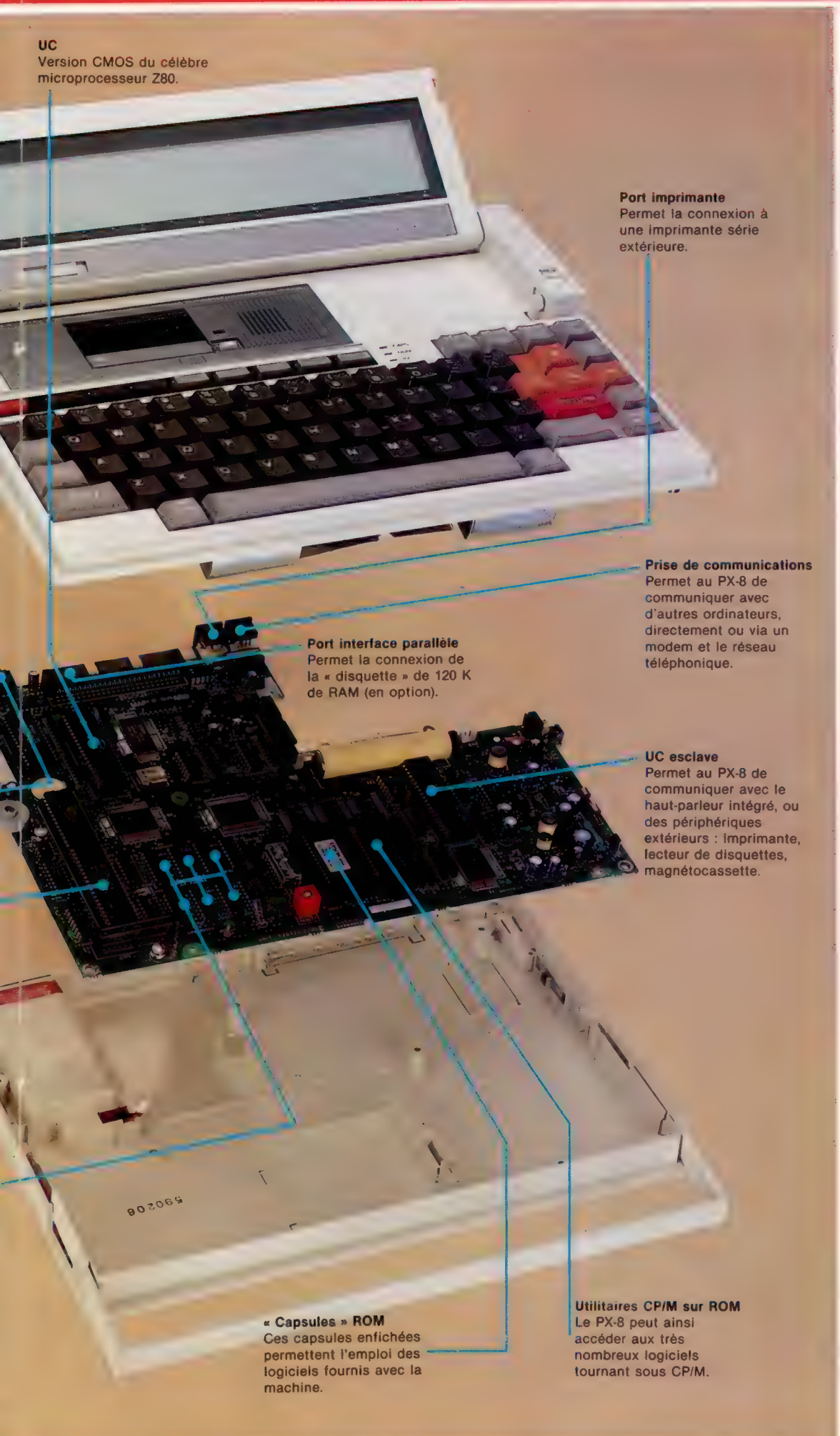
Sortie haut-parleur
Permet de connecter un haut-parleur extérieur au PX-8 afin d'obtenir un meilleur son.

Prise A/N
Pour connecter divers instruments de mesure, tels que des voltmètres.

Prise lecteur de code-barres
Le PX-8 peut ainsi être utilisé pour la gestion des stocks ou l'établissement des prix.

Sous-UC 7508
Convertit les signaux analogiques reçus par la prise A/N en signaux numériques.

RAM
Le PX-8 comporte 64 K de RAM, afin de pouvoir tourner sous CP/M. Une pile annexe permet la conservation des données, lorsque la machine est hors tension.



Chris Stevens

PX-8 EPSON

PRIX

★ ★ ★ ★

DIMENSIONS

297 × 216 × 48 mm.

UC

UC de type CMOS, compatible Z80, 2,4 MHz.

MÉMOIRE

64 K de RAM, 32 K de ROM et 6 K de RAM vidéo.

ÉCRAN

Texte : 8 lignes de 80 caractères.
Graphisme : 480 × 64 pixels.

INTERFACES

RS232, série, lecteur de code-barres, entrée analogique.

LANGAGE DISPONIBLE

BASIC Microsoft enrichi, tournant sous CP/M.

CLAVIER

72 touches de type machine à écrire, au format AZERTY, dont des touches curseur et 5 touches de fonction programmables. 12 touches peuvent servir à la création d'un pavé numérique.

DOCUMENTATION

Deux épais manuels : l'un de conduite des opérations, l'autre de programmation BASIC. Tous deux sont exhaustifs et très bien rédigés.

POINTS FORTS

Écran très large (80 caractères), ce qui facilite la lecture et la compréhension ; logiciels sur ROM, ce qui simplifie les problèmes de chargement ; extension du système très aisée.

POINTS FAIBLES

L'écran n'affiche que 8 lignes, et il est assez lent ; CP/M n'est pas un système d'exploitation très facile d'emploi, surtout pour le débutant.

Double commande

Nous avons précédemment écrit le logiciel servant à commander une voiture Lego munie d'un moteur. Nous utilisons maintenant ces principes pour commander une voiture munie de deux moteurs.

Si nous utilisons deux moteurs d'égale puissance pour déplacer un véhicule, nous pouvons commander par ordinateur toutes les directions en combinant les mouvements avant et arrière de chacun des moteurs. Il existe en fait deux méthodes pour faire tourner un véhicule à deux moteurs ; la première consiste simplement à arrêter l'un des deux moteurs, tout en faisant tourner l'autre. Le véhicule décrit un arc de cercle autour de la de(s) roue(s) immobile(s). La seconde méthode consiste à faire tourner un moteur vers l'arrière, et l'autre vers l'avant. La manœuvrabilité du véhicule, qui en tournant pivote autour de son axe central, est améliorée.

Nous pouvons commander chaque moteur de façon bidirectionnelle en utilisant les quatre sorties rouges du boîtier (basse tension) et en connectant le moteur de droite aux bornes 0 et 1, et le moteur de gauche aux bornes 2 et 3 (positives et négatives respectivement).

En plaçant le nombre approprié dans le registre de données, nous pouvons maintenant déplacer le véhicule vers l'avant ou vers l'arrière, ou le faire tourner vers la gauche ou vers la droite. Le moteur de droite (RH) avancera si la ligne 0 est au niveau élevé et si la ligne 1 est au niveau bas, et reculera dans la situation inverse. De même, le moteur de gauche (LH) avancera si la ligne 2 est au niveau élevé et si la ligne 3 est au niveau bas, etc. En combinant ces mouvements, nous pouvons commander le mouvement global du véhicule :

Mouvement du véhicule	Moteur gauche	Moteur droite	Configuration de bits	Nombre des DATREG
ARRET	ARRET	ARRÊT	'000'	0
AVANT	AVANT	AVANT	'010'	5
ARRIERE	ARRIERE	ARRIERE	'101'	10
PIVOT GAUCHE	AVANT	ARRIERE	'011'	6
PIVOT DROITE	ARRIERE	AVANT	'100'	9
ARC GAUCHE	AVANT	ARRET	'010'	4
ARC DROITE	ARRET	AVANT	'001'	1

Le programme suivant nous permet de commander le véhicule directement au clavier en utilisant T pour avancer, B pour reculer, F pour tourner à gauche, et H pour tourner à droite. Si aucune touche n'est enfoncée, le véhicule s'arrête.

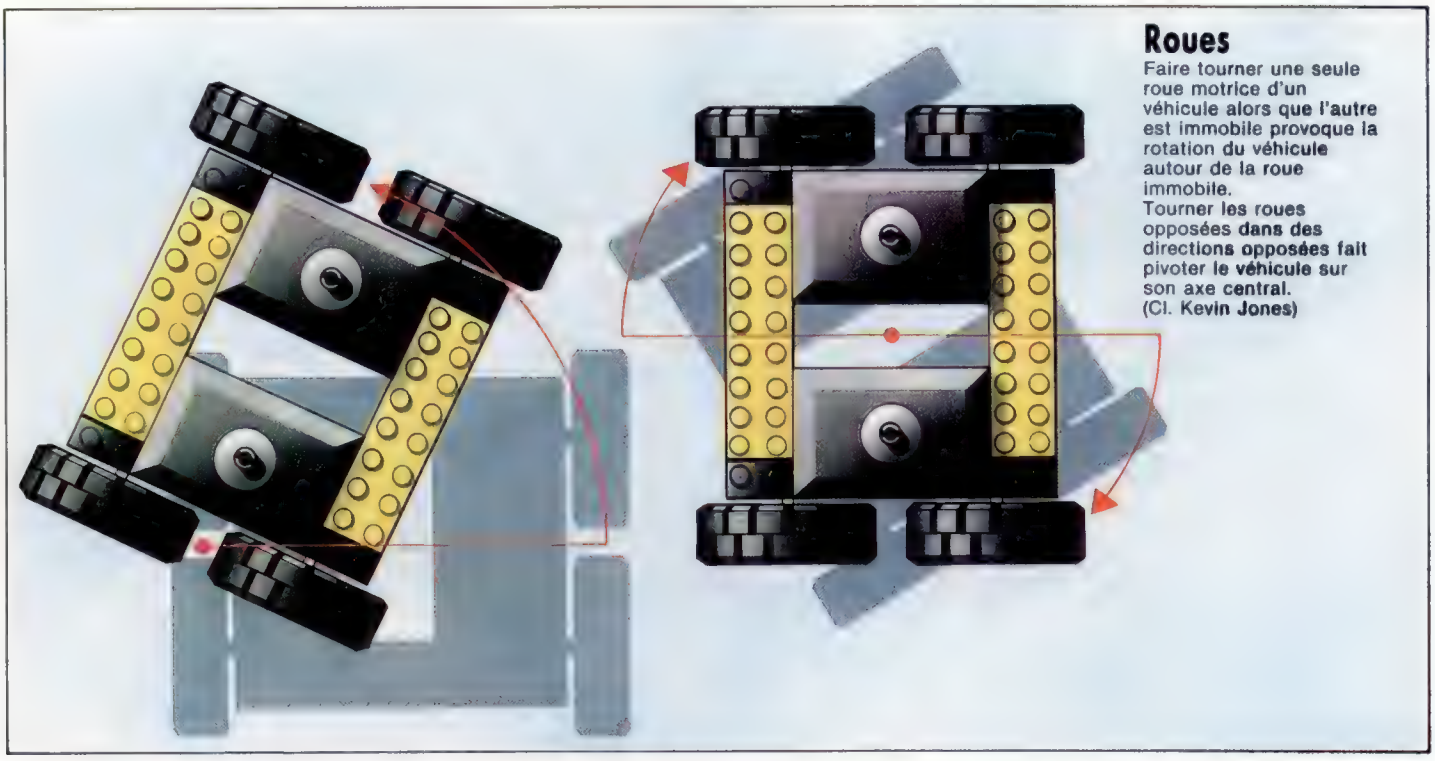
Tortue tandem
Après avoir compris comment faire avancer et reculer un véhicule Lego, nous pouvons maintenant en réunir deux. Les moteurs peuvent être interrompus individuellement, ce qui accroît leur manœuvrabilité. (Cl. Ian McKinnell).

BBC MICRO

```
 10 REM DEUX MOTEURS BBC
 20 DDR=&FE62:DATREG=&FE60
 30 ?DDR=255
 40 REPEAT
 50 A$= INKEY$(10)
 60 PROCtest_clavier
 70 UNTIL A$="X"
 80 ?DATREG=0
 90 END
1000 DEF PROCtest_clavier
1010 IF A$=""THEN?DATREG=0
1020 IF INKEY(-36)=-1 THEN?DATREG=5
1030 IF INKEY(-101)=-1 THEN?DATREG=10
1040 IF INKEY(-68)=-1 THEN?DATREG=6
1050 IF INKEY(-85)=-1 THEN?DATREG=9
1060 ENDPROC
```

COMMODORE 64

```
 10 REM DEUX MOTEURS CBM 64
 20 DDR = 56579:DATREG=56577
 25 POKE650,128:REM REPEAT MODE CLE
 30 POKE DDR, 255
 40 GET A$
 50 GOSUB1000:GOTO 70
 60 POKE DATREG,0
 70 IF A$<>"X"THEN FOR I=1 TO 100:NEXT:GOTO 40
 80 POKE DATREG,0
 90 END
1000 REM SOUS-PROGRAMME TEST ENTREE
1005 IFA$="" THEN POKE DATREG,0
1010 IFA$="T" THEN POKE DATREG,5
1020 IFA$="B" THEN POKE DATREG,10
1030 IFA$="F" THEN POKE DATREG,6
1040 IFA$="H" THEN POKE DATREG,9
1050 RETURN
```



Roues
Faire tourner une seule roue motrice d'un véhicule alors que l'autre est immobile provoque la rotation du véhicule autour de la roue immobile.
Tourner les roues opposées dans des directions opposées fait pivoter le véhicule sur son axe central.
(Cl. Kevin Jones)

Dans chaque version du programme, le véhicule ne se déplace que lorsqu'une touche est appuyée. Dès que la touche est relâchée, on met les moteurs hors tension en plaçant un zéro dans le registre de données. On peut quitter le programme en appuyant sur la touche X.

La version du Commodore 64 met immédiatement en fonction la répétition automatique des touches, afin qu'elle envoie de façon continue des caractères au tampon clavier lorsqu'une touche est maintenue enfoncée. Il est malheureusement impossible de lire directement le clavier. La rapidité de réponse peut être améliorée en effaçant le tampon du clavier juste avant de le lire. Vous devez pour ce faire insérer la ligne suivante dans la version Commodore du programme.

```
35 GET J$:IF J$ < >«»THEN 35
```

De plus, l'instruction GOTO de la ligne 70 devient GOTO 35.

La vitesse de répétition d'une touche, lorsqu'elle est maintenue enfoncée, peut causer un problème dans les deux versions du programme. Si la boucle du programme principal est exécutée plus rapidement que la période de répétition de la touche, la routine tirera la conclusion qu'aucune touche n'est pressée lorsqu'elle testera le clavier. Cela donnera une alternance rapide de mise sous tension du moteur, puisque la sortie passe rapidement de zéro à la valeur correspondant à la direction choisie. Dans chacune des versions du programme, ce problème a été évité en ajoutant des instructions qui ralentissent le temps d'exécution de la boucle principale du programme. Dans la version du Commodore 64, une courte boucle de retard a été ajoutée à la ligne 60. Les valeurs de ces retards furent trouvées de façon empirique, et dépendent du temps d'exécution de la routine.

Maintenant que nous commandons les mouvements de notre véhicule, il est intéressant de concevoir un programme qui mémorisera une séquence de mouvements et les répétera. Pour ce faire, nous devons utiliser un tableau à deux dimensions qui enregistre la direction et le temps nécessaire pour effectuer chaque manœuvre. La première partie d'un tel programme est identique aux instructions déjà données, mais la seconde partie répète les données stockées. Les données sont stockées dans un tableau DR(), où DR(C,1) stocke la direction et où DR(C,2) mémorise le temps requis par chaque mouvement. Un nouvel élément du tableau est utilisé chaque fois qu'une nouvelle direction est sélectionnée. Cette condition est indiquée par un changement du contenu du registre de données. Un compteur, C, sert à retracer les éléments du tableau.

BBC MICRO

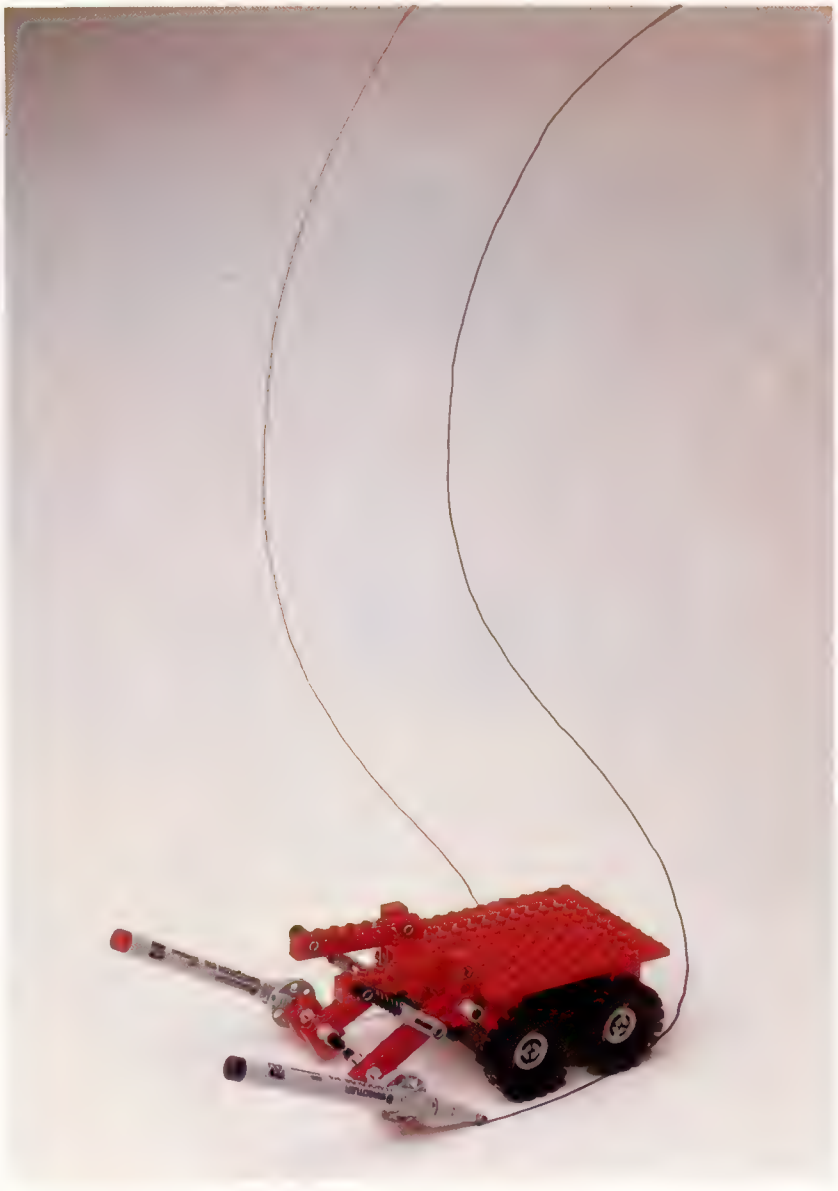
```
1000 REM MÉMOIRE DES MOUVEMENTS BBC
1010 DDR=&FE62:DATREG=&FE60
1020 DIM DR(100,2)
1030 ?DDR=255:C=1: REM INITIALISE LE COMPTEUR
1040 REPEAT
1050 A$=INKEY$(10)
1060 PROCtest_clavier
1070 UNTIL A$=«X»
1080 ?DATREG=0
1090 DR(C-1,2)=TEMPS
1100 REPEAT A$=GET$
1110 UNTIL A$=«C»
1120 REM RÉPÈTE LES DONNÉES
1130 FOR I=1 TO C
1140 ?DATREG=DR(I,1)
1150 TEMPS=0
```

```
1160 REPEAT UNTIL TEMPS >=DR(I,2)
1170 NEXT I
1180 END
1190 :
1200 DEF PROCtest_clavier
1210 IFA$=«»THEN?DATREG=0
1220 IF INKEY(-36)=-1 THEN?DATREG=5
1230 IF INKEY(-101)=-1 THEN?DATREG=10
1240 IF INKEY(-68)=-1 THEN?DATREG=6
1250 IF INKEY(-85)=-1 THEN?DATREG=9
1260 PT=?DATREG
1270 IF PT<>DR(C-1,1)THEN PROCajoute_données
1280 ENDPROC
1290 :
1300 DEF PROCajoute_données
1310 DR(C-1,2)=TIME: REM STOCKE DERNIER TEMPS
1320 TIME=0:REM DEMARRE NOUVEAU TEMPS
1330 DR(C,1)=PT:REM STOCKE STATUT DU PORT
1340 C=C+1:REM INCREMENTE LE COMPTEUR
1350 ENDPROC
```

COMMODORE 64

```
 10 REM MÉMOIRE DE MOUVEMENTS CBM 64
 15 DIM DR(100,2): REM TABLEAU DE DIRECTION
 20 DDR=56579:DATREG=56577
 25 POKE 650,128 : REM SOLLICITE MODE RÉPÉTITION DE TOUCHES
 30 POKE DDR,255: REM TOUTES SORTIES
 35 C=1: REM INITIALISE LE COMPTEUR
 40 GET A$
 50 GOSUB1000: REM TESTE L'ENTRÉE
 70 IF A$<>«X»THEN FOR I=1 TO 200:NEXT:GOTO 40
 80 POKE DATREG,0:REM ARRÊT
 85 DR(C-1,2)=TI-T: REM ENTRE DERNIER TEMPS
 90 STOP: REM TAPER «CONT» POUR CONTINUER
 95 REM RÉPÈTE LES DONNÉES
100 FOR I=1 TO C
110 POKE DATREG,DR(I,1)
120 T=TI
130 IF (TI-T)< DR(1,2)THEN 130
140 NEXT
150 END
999 :
1000 REM S/P DE TEST D'ENTRÉE
1005 IF A$=«»THEN POKE DATREG,0
1010 IF A$=«T»THEN POKE DATREG,5
1020 IF A$=«B»THEN POKE DATREG,10
1030 IF A$=«F»THEN POKE DATREG,6
1040 IF A$=«H»THEN POKE DATREG,9
1045 PT=PEEK(DATREG)
1050 IF PT<>DR(C-1,1)THEN GOSUB1500
1498 RETURN
1499 :
1500 REM AJOUTE DES DONNEES AU TABLEAU
1510 DR(C-1,2)=TI-T: REM AJOUTE DERNIER TEMPS
1520 T=TI: REM LIRE NOUVEAU TEMPS
1530 DR(C,1)=PT: REM ENTRE CONTENU ACTUEL DU PORT
1540 C=C+1: REM INCRÉMENTE LE COMPTEUR
1999 RETURN
```

Ce programme permet à l'utilisateur de déplacer le véhicule sous la commande de l'ordinateur. Puisque chaque mouvement est donné par une direction et par un intervalle de temps, toute erreur de synchronisation dans les mouvements en produira lors de la répétition des mouvements. Telle est la programmation en temps réel.



Exercices

Maintenant que nous pouvons commander les déplacements d'un véhicule dans les quatre directions, il existe plusieurs possibilités d'exercices de programmation. Vous avez sans doute de nombreuses idées; voici quelques suggestions :

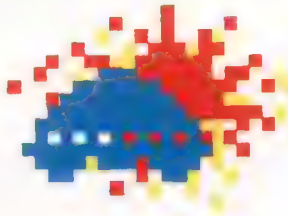
1. Essayez de calibrer votre véhicule. Quel nombre doit être placé dans le registre de données pour déplacer le véhicule de 1 m vers l'avant ou vers l'arrière, ou le faire tourner de 90° ?

2. Concevez un parcours d'obstacles et, en vous basant sur les programmes donnés, écrivez un programme qui vous permette d'apprendre à votre véhicule à négocier le parcours sous le contrôle du clavier. Une fois que vous avez guidé le véhicule tout au long du parcours, le programme devrait entrer en jeu et guider le véhicule jusqu'à son point de départ, en reprenant le même chemin.

3. Connectez les quatre interrupteurs au boîtier tampon, ce qui vous permettra de diriger le véhicule de l'extérieur, à partir du port utilisateur.

Mouvements mémorisés
Il est assez simple d'écrire un programme de commande d'un véhicule qui accepte de donner les directions à partir du clavier. Il n'est pas beaucoup plus difficile d'augmenter le programme, afin qu'il stocke les commandes de l'opérateur, et qu'il répète les mouvements du véhicule ainsi définis. La comparaison du parcours initial avec le parcours reproduit illustre les difficultés que l'on rencontre dans le monde réel : l'ordinateur utilise des nombres et des durées exacts sur un modèle qui représente un univers parfait, qui ne tient pas compte de l'inertie, du frottement, des surfaces irrégulières et des imprécisions techniques.

Les envahisseurs



Aucun jeu vidéo n'a su enflammer l'imagination du public comme Space Invaders. Simple mais passionnant, il fut rapidement adapté à divers micros. Voyons la version originale, due à Atarisoft.

Tout micro-ordinateur digne de ce nom possède au moins une version de Space Invaders. C'est même un nom si connu qu'on l'utilise indifféremment pour tout jeu vidéo qui met en scène une série de combats galactiques. Dès son apparition en 1978, Space Invaders connut un succès phénoménal, qui prit très vite des allures d'épidémie : bien des parents se demandèrent avec inquiétude si leurs enfants ne finiraient pas par passer tout leur temps dans les salles de jeux et les cafés, réduits à l'état de zombies... Ils ne voyaient pas que l'avenir était en marche.

Il n'est pas exagéré de dire que Space Invaders a complètement bouleversé l'idée que la société se faisait des ordinateurs. Jusqu'alors, et dans le meilleur des cas, ils ne faisaient guère penser qu'au sinistre HAL, l'inquiétant mutant électronique du film *2001, l'Odyssée de l'espace*. Le jeu fut aussi le grand précurseur de ce qu'on appelle en anglais les *Shoot-em-up games,* (« tuez-les tous »), dans lesquels un héros (beaucoup plus rarement une héroïne...) doit faire face à d'innombrables vagues d'assaillants venus de l'espace, plus abominables les uns que les autres. Sa seule arme : un manche à balai fiable et un index infatigable, pour appuyer sur le bouton de mise à feu... Bien entendu, Space Invaders a quelque peu vieilli ; il reste assez simpliste dans sa conception. Pourtant, aucun des milliers de jeux vidéo sortis depuis n'est parvenu, comme il l'a fait si aisément, au rang de véritable légende.

Son principe : le joueur contrôle une base de canons laser placée en bas de l'écran, et la déplace horizontalement pour tirer sur les envahisseurs extraterrestres qui, menaçants, descendent peu à peu du haut de l'écran pour attaquer la Terre. Chaque fois que la base est touchée ou qu'un vaisseau ennemi parvient à la surface de la planète, une « vie » est perdue.

Par rapport à la version originale, les adaptations sur micro-ordinateur comportent certaines différences. Les extraterrestres ne jaillissent plus de nulle part, mais d'une fusée mère placée en haut et à gauche de l'écran. Le joueur ne peut plus se dissimuler derrière des barrières défensives, et l'adversaire doit parcourir une distance bien plus réduite. Le graphisme est beaucoup plus soigné et complexe.

Mais il faut admettre qu'une chose essentielle, pourtant, n'a pas changé : c'est l'obsédant battement de cœur qu'on entend, de plus en plus fort, à mesure que les envahisseurs descendent. Il ne manque jamais de provoquer une violente décharge d'adrénaline, ce qui est sans doute l'une des raisons fondamentales du succès énorme de ce logiciel.

Autre détail commun à toutes les versions, le « bonus » aléatoire offert au joueur quand il parvient à détruire l'une des soucoupes volantes qui traversent de temps en temps l'écran, de la gauche vers la droite.

Six ans après son apparition, Space Invaders a gardé tout son attrait. Il reste aussi passionnant — et éprouvant ! — qu'au début, malgré la parution de programmes infiniment plus sophistiqués, et il est, d'ores et déjà, un véritable classique.

Space Invaders : pour tous les ordinateurs Atari.
Éditeurs : Atari Corporation.
Auteurs : Atarisoft.
Manche à balai : indispensable !
Format : cartouche.

Liz Dixon

Précieux pointeurs

Nous allons regarder de près la sauvegarde et le chargement d'un jeu de caractères redéfini dans une zone spécifique du Commodore 64... Toujours des progrès dans les programmes.

Il est temps de comparer les trois versions de nos programmes de définition de caractères, et de les améliorer. La version du Commodore a été écrite en premier, car c'est la plus difficile à programmer. Cette version a été ensuite transcrite, pratiquement ligne par ligne, pour les deux autres machines. A cause de cette transcription et à cause du manque de place, le formatage de l'écran est rudimentaire; il n'est pas fait usage de la couleur, du son ou du graphique haute résolution. On peut donc apporter beaucoup d'améliorations dans ce domaine, mais nous n'aborderons pas cette question ici.

Laissant de côté les questions d'efficacité du programme (qui n'ont pas une importance vitale, puisqu'il n'y a pas de tâches dépendant de la vitesse du code), nous nous consacrerons à l'interface utilisateur : les instructions, l'aide, les touches de commande et les diverses facilités.

Le choix des touches de commande peut être amélioré. Sur le BBC Micro et sur le Spectrum, le curseur est déplacé dans une fenêtre par les touches habituelles de contrôle du curseur, alors que, sur le Commodore, les touches non préfixées de fonction sont utilisées. Cela permet une programmation très pratique pour la version Commodore, du fait que les codes ASCII des huit touches de fonction se suivent entre 133 et 140, même si la disposition des touches de fonction n'est pas très ergonomique. Ces dernières ne comportent pas la répétition automatique. Cet inconvénient peut être remédié par POKE 650,128.

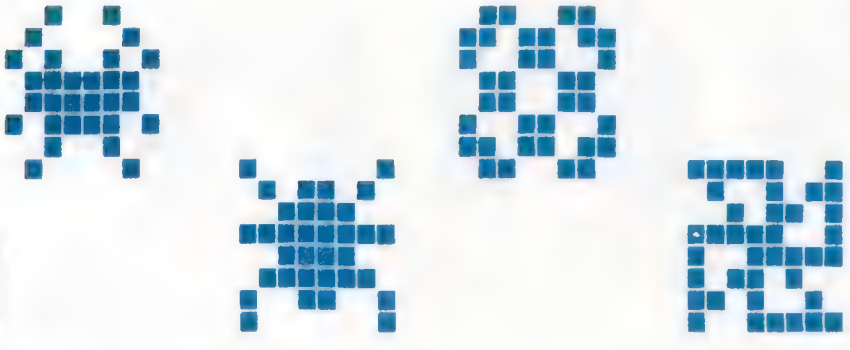
Une autre amélioration possible relève du choix de la stratégie de déplacement du curseur. Tel qu'il est écrit, le programme refuse comme illégale toute commande qui ferait sortir le curseur des limites de l'écran. L'alternative est de retenir le mode enroulement. Cela peut se faire de plusieurs manières : si le curseur est admis à dépasser le bas de l'écran, il peut s'enrouler sur l'écran et réapparaître en haut, et vice versa. L'enroulement horizontal se fera de manière similaire. Cela peut se programmer facilement, mais suppose davantage de code que pour les simples tests du sous-programme 3500.

Les commandes indiquées constituent le minimum et pourraient bien être étoffées. Pour les trois versions, il serait utile de pouvoir copier la définition d'un caractère pour un autre caractère, de sorte que CHR$(N) et CHR$(N+1) représentent tous les deux le même caractère, par exemple. Une recopie d'écran d'un nouveau jeu de caractères peut être également utile, aussi une option d'impression pourrait également être ajoutée.

Un problème propre au BASIC du Commodore est la commande SAVE, qui ne concerne que la zone du programme BASIC, alors que les autres machines permettent également de spécifier la zone de la mémoire à sauvegarder. Cependant, la commande LOAD (chargement) permet de charger des fichiers sur n'importe quelle zone. Si nous parvenons à résoudre le problème de SAVE, nous serons à même de sauvegarder et de charger le nouveau jeu de caractères.

SAVE est pointée sur la zone du programme BASIC du Commodore par deux pointeurs d'adresse, TXTTAB (aux positions 43 et 44), et VARTAB (aux positions 45 et 46). Le premier pointe sur le début de la zone du programme BASIC (généralement aux adresses 2048 et suivantes), le deuxième pointe sur le début de la zone des variables BASIC. Du fait que la zone des variables BASIC commence là où se termine celle du programme, VARTAB pointe effectivement sur la fin de la zone du programme BASIC. Si nous modifions ces pointeurs de sorte qu'ils indiquent le début et la fin du nouveau jeu de caractères, et si nous lançons ensuite une commande de sauvegarde SAVE, nous aurons résolu le problème.

Mais, avant cela, il nous faut revoir la position du jeu de caractères lui-même. Le sous-programme de la ligne 61000 copie le jeu de caractères de la ROM sur un bloc de 2 K de RAM commençant en 14336. La ligne 50 positionne le pointeur de dessus de mémoire sous ce bloc, afin d'empêcher l'écriture BASIC de le recouvrir. De sorte que, sous prétexte de protéger 2 K de mémoire, on réduit la mémoire utilisateur des deux tiers. Cela ne pose pas de problème lors de l'exploitation du programme générateur de caractères, mais ce sera une source potentielle de difficultés, lorsque nous voudrons charger le nouveau jeu de caractères à cette adresse pour qu'il y soit utilisé par un programme d'applications, lequel suppose plus de 12 K de mémoire utilisa-



Ian McKinnell

teur. Malheureusement, le système d'exploitation ne permet pas de placer un jeu de caractères plus haut dans la mémoire. Si cela avait été possible, nous l'aurions placé sur les 2 K les plus hauts de la mémoire utilisateur, ou dans la zone spéciale du programme située à partir de 49152. La solution est donc, au contraire, de placer ce jeu de caractères le plus bas possible et de déplacer le BASIC au-dessus de lui! Cela peut être fait en modifiant le contenu des pointeurs TXTTAB, mais ne peut être obtenu de l'intérieur d'un programme BASIC. Il faudra donc le faire avant que le programme générateur de caractères soit chargé en mémoire.

La démarche à suivre est la suivante :

1. Charger et exécuter le programme 1 (LOAD et RUN). Cela affiche les commandes de relocation mémoire nécessaires, de sorte que vous pouvez les exécuter en mode direct, en appuyant simplement sur retour-chariot.
2. Charger le programme générateur de caractères (LOAD); apporter les modifications suivantes :

```
61100 CGEN=53248:NCGEN=2048
61500 POKE PO,(PEEK(PO)AND240)OR2
```

et supprimer la ligne 50.

3. Sauvegarder cette nouvelle version.
4. Charger et exécuter (LOAD et RUN) le générateur de caractères tout comme avant.
5. Lorsque vous en avez fini avec ce dernier, chargez et exécutez le programme 2 (LOAD et RUN). Tout comme pour le programme 1, cela affiche les commandes à exécuter.
6. Les pointeurs TXTTAB et VARTAB ont été positionnés par le programme 2, aussi SAVE«nom du fichier» sauvegarde la totalité de la zone du jeu de caractères de 2 K, située entre 2048 et 4097. A l'avenir, pour exécuter le programme générateur de caractères, vous devrez suivre cette démarche, à l'exception de la deuxième étape.

Lorsque vous voulez restituer le jeu de caractères, il vous faut charger et exécuter le programme 1 pour déplacer le BASIC vers le haut de la mémoire, et ensuite charger le jeu de caractères de la sorte :

```
LOAD«nom du fichier»,NP,1
```

où NP (numéro du périphérique) est égal à 1 pour l'utilisation sur cassette, et à 8 pour un lecteur. Le 1, à la fin de la commande, est connu sous le nom d'« adresse secondaire » et constitue le moyen propre à Commodore d'envoyer des paramètres de commande aux unités périphériques. Il signifie ici que le fichier doit être chargé sur l'emplacement mémoire à partir duquel il a été sauvegardé, plutôt que d'être dirigé par le pointeur TXTTAB dans la zone du programme BASIC courant. Cela est possible quand un fichier est sauvegardé, car le système d'exploitation sauvegarde l'adresse du début en RAM, comme le premier élément de données du fichier. Lorsque vous utilisez la commande de base LOAD, l'adresse du début de fichier est ignorée au profit de l'adresse pointée par TXTTAB. Une fois que vous avez déplacé le BASIC et chargé le nouveau jeu de caractères, vous devez faire pointer le système d'exploitation sur le nouveau jeu de caractères. Cela est expliqué dans la table ci-dessus, et mis en œuvre dans la nouvelle version de la ligne 61500.

valeur de X =	bits 3,2,1,0	position pointée
0	0000	0
2	0010	2048
4	0100	4096
6	0110	6144
8	1000	8192
10	1010	10240
12	1110	12288
14	1110	14336

Table des valeurs
La commande POKE53272,(PEEK(53272)AND240)OR X contraint le composant d'affichage à se pointer sur la zone de mémoire vive comportant le jeu de caractères redéfini. La table indique les valeurs de X qui déterminent l'adresse de début de bloc de mémoire vive.

```
199 REM ********************************
200 REM *          PROGRAMME 1           *
201 REM *      EXÉCUTE CE PROGRAMME      *
202 REM *        PUIS FAIT 2 RC          *
203 REM *   CELA DÉPLACE LE BASIC EN 4096 *
204 REM ********************************
300 PRINT CHR$(147):PRINT:PRINT
400 PRINT"POKE 43,0:POKE 44,16:POKE 45,3:POKE 46,16"
500 PRINT"POKE 4096,0:POKE 4097,0:POKE 4098,0:CLR:NEW"
600 PRINT CHR$(19)
```

```
199 REM ********************************
200 REM *          PROGRAMME 2           *
201 REM *      EXÉCUTE CE PROGRAMME      *
202 REM *        PUIS FAIT 2 RC          *
203 REM *  CELA RE-INITIALISE PTRS DU BASIC *
204 REM ********************************
300 PRINT CHR$(147):PRINT:PRINT
400 PRINT"POKE 43,0:POKE 44,8:POKE 45,1:POKE 46,16"
500 PRINT"POKE 4096,0:POKE 4097,0:POKE 4098,0:CLR"
600 PRINT CHR$(19)
```

Correspondances

L'adressage indexé sur le processeur 6809 sert ici à effectuer des opérations arithmétiques simples sur les valeurs contenues dans les registres d'index.

Précédemment, nous avons jeté un œil sur l'adressage indexé du processeur 6809. En mode indexé, l'adresse effective spécifiée, par exemple, par OFFSET,X est formée par la somme du décalé (qui peut être une constante ou le contenu d'un emplacement mémoire) et la valeur en cours contenue dans le registre d'index spécifié (dans ce cas, le registre X). Nous avons vu que, dans certaines situations, le décalé pouvait être zéro, auquel cas nous pouvons écrire ,X (cependant que 0,X ne serait pas faux). Dans des cas particuliers, un des accumulateurs A, B ou D peut être utilisé comme décalé (B,X). Nous avons vu comment l'utilisation la plus simple de l'indexation — parcourir une table des valeurs — est facilitée par l'emploi des modes d'auto-incrémentation et décrémentation.

A présent, considérons comment l'adressage indexé peut être utilisé pour des opérations simples portant sur des valeurs dans le registre d'index à l'aide de l'instruction LEA (Load Effective Address/charge l'adresse effective). Les instructions arithmétiques normales ne fonctionneront pas sur les valeurs des registres autres que les accumulateurs. Bien qu'il soit possible de transférer le contenu du registre d'index dans l'accumulateur D, effectuer l'opération puis retransférer le résultat sont des procédures lourdes et lentes. L'instruction LEA (qui ne peut être appliquée qu'aux registres X, Y, S et U) effectuera tous les calculs d'adresses nécessaires, puis chargera la valeur d'adresse effective. Normalement, le contenu d'une adresse effective devrait être chargé, aussi est-ce une possibilité utile.

Prenons un autre exemple. L'instruction :

```
LEAX -1,X
```

calculera l'adresse effective comme la somme de —1 et du contenu du registre X. Cette adresse est ensuite chargée en X, ce qui décrémente en effet la valeur contenue dans ce registre. Ce n'est pas le seul usage de cette instruction; elle peut aussi servir, par exemple, à effectuer une fois un calcul d'adresse et à sauvegarder le résultat, plutôt que de faire ce même calcul plusieurs fois.

Il est aussi possible de faire un certain nombre d'opérations arithmétiques sur le registre X à l'aide de l'instruction ABX (Add B to X/additionner B à X) qui fait l'addition non signée du contenu de B avec le contenu de X. Mais cette instruction n'est pas aussi fréquente que LEA.

Un sous-programme est une partie autonome de programme, appelée à partir du programme principal (ou d'un autre sous-programme) pour accomplir une tâche spécifique. Une fois qu'elle a accompli sa fonction, le contrôle est automatiquement transféré au programme qui l'a appelée, à l'instruction suivant immédiatement l'appel de sous-programme. Il y a principalement trois raisons d'utiliser des sous-programmes :

1. Éviter de réécrire plusieurs fois la même séquence d'instructions. Il est plus pratique de l'écrire une fois en sous-programme et de l'appeler quand cela est nécessaire.
2. Constituer une bibliothèque de sous-programmes courants qui puissent être utilisés dans différents programmes.
3. Subdiviser un programme en parties plus petites et plus faciles à gérer.

Sous-programme

L'essentiel est de se souvenir, lorsqu'on utilise des sous-programmes en langage d'assemblage, que le programme, qui appelle, et le sous-programme utilisent les mêmes registres. L'une des erreurs les plus communes en programmation langage machine se produit lorsque, après avoir stocké une valeur dans l'un des registres, un programme appelle un sous-programme et, à son retour, trouve que le contenu de ce registre a été modifié par le sous-programme. C'est pourquoi il est vital de connaître et de bien documenter les registres qu'utilise un sous-programme. Il est particulièrement essentiel de sauvegarder le contenu des registres en cours d'utilisation au moment de l'appel au sous-programme, et de restaurer ces contenus lorsque le contrôle revient du sous-programme.

Plus tard, nous verrons comment on se sert des piles, à la fois pour sauvegarder de telles données et pour faire passer des valeurs et des adresses (paramètres) au sous-programme. Pour l'instant, nous supposerons que le sous-programme utilise les mêmes données que le programme qui appelle (variables globales), et que toutes les autres valeurs dont il a besoin se trouvent dans les registres. Un appel de sous-programme est fait à l'aide de l'une de ces instructions :

- BSR : Branchement à sous-programme.
- JSR : saut (Jump) à sous-programme.

La commande BSR cause un branchement relatif — elle trouve le sous-programme à un certain décalé de la valeur courante du compteur de programme. Cette instruction est normalement utilisée pour les sous-programmes écrits à l'intérieur du programme.

L'instruction JSR appelle un sous-programme à une certaine adresse spécifiée. Elle est utilisée pour un sous-programme contenu en ROM, ou pour un programme de bibliothèque qui occupe toujours la même position en mémoire — des parties du système d'exploitation de disque (DOS), par exemple.

Lorsque le processeur rencontre une instruction BSR ou JSR, la valeur en cours du compteur de programme est entrée (PUSH) sur la pile à l'aide du registre S (pointeur de pile). Si votre sous-programme utilise le registre S pour autre chose que pour un nouvel appel de sous-programme, vous devez vous assurer qu'il revient à la valeur correcte. L'adresse de sous-programme est calculée (dans le cas de BSR) et chargée dans le compteur de programme. Ainsi, la prochaine instruction à laquelle on a accès sera la première du sous-programme. Il faut donc s'assurer que le sous-programme commence par une instruction, et non par un octet de donnée.

Un sous-programme doit se terminer par une instruction RTS (ReTour de Sous-programme), dont l'effet est de sortir (PULL) l'ancienne valeur du compteur de programme de la pile. L'exécution du programme continuera alors à partir de là où on l'a laissé avant l'appel de sous-programme.

L'exemple de programme que nous donnons ici est relativement plus complexe que les précédents, mais il est rendu plus pratique par l'utilisation d'un sous-programme. Le programme cherche un tableau contenant des chaînes de longueurs inégales, et extrait une valeur associée à une chaîne particulière. Les chaînes sont sous la forme habituelle : commençant par un octet indiquant leur longueur, suivies par les caractères qui les composent, et terminées par une adresse à 16 bits associée à la chaîne.

La fin du tableau est marquée par une chaîne de longueur nulle — autrement dit, il y a un zéro à l'emplacement de l'octet de longueur. Nous supposerons que l'adresse de début du tableau est contenue dans $10, et l'adresse de la chaîne dont nous devons chercher la correspondance est contenue en $12. Si son double est trouvé dans le tableau, alors l'adresse correspondante doit se trouver en $14. Si on ne trouve pas la chaîne, $12 et $14 doivent tous deux être mis à zéro.

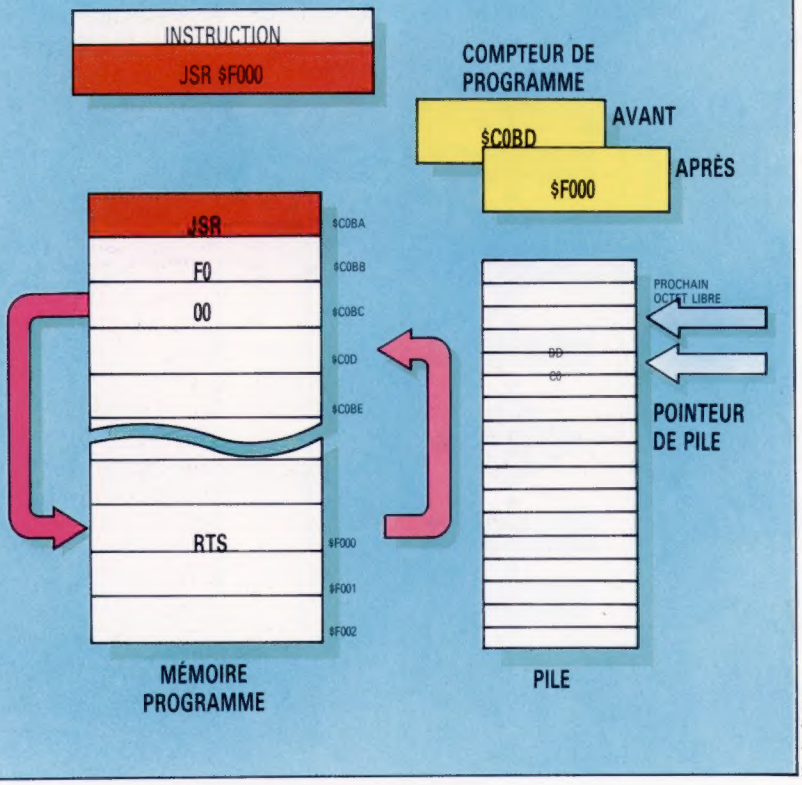
Correspondance de chaînes

La correspondance de chaîne est une tâche qui revient très souvent — surtout lorsqu'il s'agit d'accès de variable chaîne d'un interpréteur BASIC : chaque identificateur (ou nom de variable) doit être remplacé par l'adresse dans laquelle est stockée la valeur de cette variable.

Le problème se scinde en deux parties : il faut parcourir le tableau jusqu'à trouver soit la chaîne que nous cherchons, soit la fin du tableau. A chaque étape de la recherche, nous devons comparer deux chaînes (celle que nous cherchons et celle de la position en cours dans le tableau) pour voir

Sauteur de pile

Le « saut et retour », présent dans un appel de sous-programme, est réussi en sauvegardant la valeur du compteur de programme, puis en la remplaçant avec l'adresse de l'appel de sous-programme et, enfin, en remettant à son état initial le compteur de programme. La pile est dans une zone de mémoire utilisée par le processeur pour sauvegarder l'adresse de retour. Le pointeur de pile est une UC de 16 bits qui contient toujours l'adresse des trois prochains octets libres de l'espace pile. Quand JSR apparaît, par exemple à l'adresse $C0BA, l'UC place automatiquement $C0BD — l'adresse d'ABX, l'instruction qui suit JSR — dans le compteur de programme. Quand JSR est exécutée, les contenus du compteur de programme sont déversés dans la pile par l'UC et $F000 est introduit dans le compteur de programme. Le sous-programme est alors exécuté jusqu'à ce que RTS soit rencontrée, quand $C0BD — l'adresse de retour — est poussée du fond de la pile dans le compteur de programme.



Kevin Jones

de quelle manière elles se correspondent. La comparaison de chaîne est un candidat évident à un sous-programme, non seulement parce qu'elle sera utilisée plus d'une fois dans le programme, mais aussi parce qu'elle nous permet de scinder le problème en sections utiles. C'est aussi un bon sous-programme qui pourra servir dans d'autres programmes.

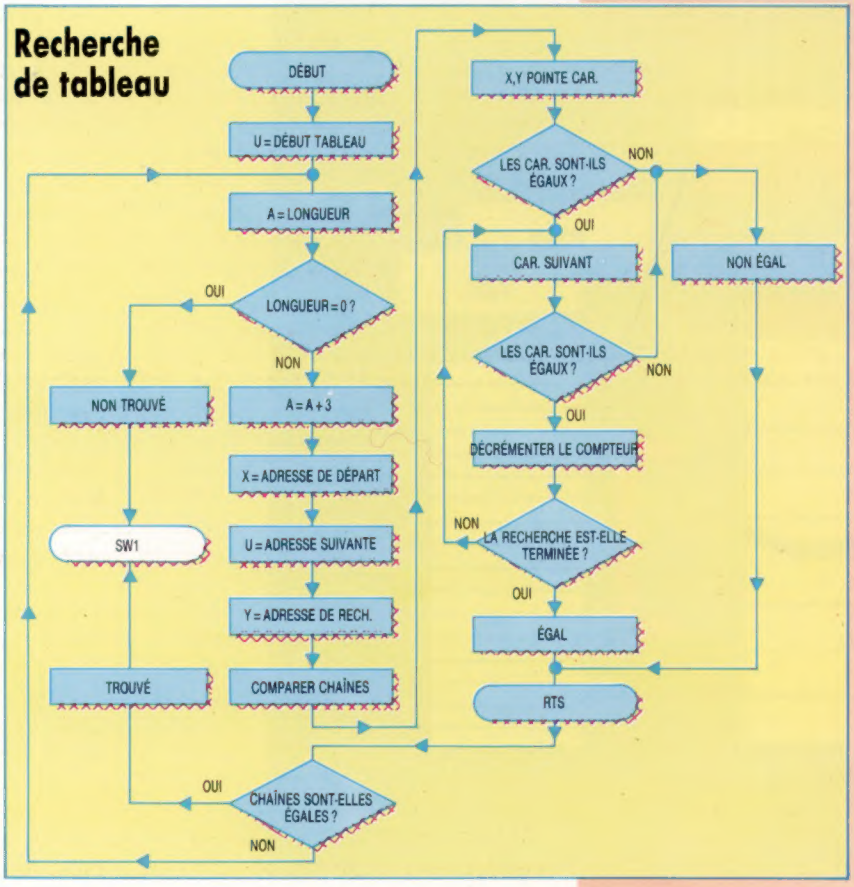
Vrai ou faux

Il est possible de faire passer un paramètre booléen (vrai ou faux) en utilisant un des drapeaux du registre de code condition, mais cela requiert la connaissance exacte de l'effet de chaque instruction sur les drapeaux. Dans notre programme, nous repasserons les valeurs à la routine d'appel comme $00 (seulement des 0) si la correspondance est trouvée, ou $FF (seulement des 1), dans le cas contraire.

Pour généraliser le sous-programme, nous ne ferons pas repasser l'adresse en cours pour une correspondance trouvée, mais nous laisserons le registre X pointer sur l'adresse où peut être trouvée l'adresse requise. Cela présente l'avantage supplémentaire que le registre X, en parcourant la chaîne octet par octet, devrait se terminer en contenant automatiquement cette information dans tous les cas.

Enfin, notre programme contient une nouvelle instruction 6809. TST (TeST) n'a d'effet sur aucun registre, mais elle affiche simplement les drapeaux suivant la valeur en cours dudit registre.

```
TABLE   EQU   $10          Sauvegarde ce qu'on
STRNG   EQU   $12          cherchait.
ADDRS   EQU   $14          Fin de programme
        ORG   $1000        principal.
        LDU   TABLE        Début de
LOOP1   LDA   U            sous-programme.
        BEQ   NTFND1       Donne longueur d'octets
        ADD   #3           et pointe X et Y pour les
        TFR   U,X          premiers caractères.
        LEAU  A,U          Si les chaînes ne sont
        LDY   STRNG        pas de la même longueur,
        BSR   COMPAR       alors aller en NOTEQ.
        TSTA               Prend caractère suivant
        BEQ   FOUND1       de la chaîne du tableau.
        BRA   LOOP1        Met début de chaîne
FOUND1  LDD   ,X           dans tableau en X.
        BRA   FINSH1       Met U pour pointer
NTFND1  LDD   #0           prochaine entrée.
FINSH1  STD   ADDRS        Y pointe début de
        SWI                chaîne de recherche.
COMPAR  LDB   ,X+          Compare les deux
        CMPB  ,Y+          chaînes.
        BNE   NOTEQ        Voit si elles sont égales.
LOOP2   LDA   ,X+          Si elles sont égales, va
        CMPA  ,Y+          en FOUND1.
        BNE   NOTEQ        Sinon prend l'entrée
        DECB               suivante de tableau.
        BGT   LOOP2        Si trouvé, X doit pointer
        CLR                l'adresse que nous
        BRA   FINSH2       voulons.
NOTEQ   LDA   #$FF         Sinon,
FINSH2  RTS                l'adresse est nulle.
        END                Retour au programme
                           d'appel.
```



Page manquante
(publicité)

Page manquante
(publicité)